Practical Private Shortest Path Computation Based on Oblivious Storage

Dong Xie*, Guanru Li*, Bin Yao*, Xuan Wei*, Xiaokui Xiao†, Yunjun Gao‡, Minyi Guo*,
*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China
E-mail: {skyprophet@, lgr150@, yaobin@cs., wx1129@, guo-my@cs.}sjtu.edu.cn
†School of Computer Engineering, Nanyang Technological University, Singapore.
Email: xkxiao@ntu.edu.sg
‡College of Computer Science, Zhejiang University, China
Email: gaoyj@zju.edu.cn

***Abstract*—As location-based services (LBSs) become popular, location-dependent queries have raised serious privacy concerns since they may disclose sensitive information in query processing. Among typical queries supported by LBSs, shortest path queries may reveal information about not only current locations of the clients, but also their potential destinations and travel plans. Unfortunately, existing methods for private shortest path computation suffer from issues of weak privacy property, low performance or poor scalability. In this paper, we aim at a strong privacy guarantee, where the adversary cannot infer almost any information about the queries, with better performance and scalability. To achieve this goal, we introduce a general system model based on the concept of Oblivious Storage (OS), which can deal with queries requiring strong privacy properties. Furthermore, we propose a new oblivious shuffle algorithm to optimize an existing OS scheme. By making trade-offs between query performance, scalability and privacy properties, we design different schemes for private shortest path computation. Eventually, we comprehensively evaluate our schemes upon real road networks in a practical environment and show their efficiency.**

***Keywords*—Data Privacy, Oblivious Storage, Shortest Path, Road Network**

I. Introduction

With the growing use of positioning system in mobile devices, an expanding market for location based services (LBSs) has been created. Clients of such services may use their smart phones to get driving directions, to retrieve facilities nearby (e.g. clinics, pharmacies and police stations) or to learn social contacts close to them. However, location-dependent queries raise serious privacy problems by revealing critical personal information on health status, shopping habits, etc. These information might be tracked and misused by service providers in many forms, such as commercial profiling, governmental surveillance and intrusive advertising. In LBSs, a shortest path query asks for the path with minimum cost between a source s and a destination t in road networks. Unfortunately, shortest path queries may disclose much more information (e.g. potential destinations and travel plans of the clients) than other LBSs queries (e.g. nearest neighbor queries).

In recent years, several solutions have been proposed to address these privacy issues. However, all of them encounter problems on weak privacy property, performance or scalability. In particular, location obfuscation based methods [1] may reveal information about query locations and shortest path composition. Oblivious graph algorithms [2] suffers from low performance and high space consumption. Currently, private information retrieval (PIR) based solutions, CI and PI [3], are the state of the arts of private shortest path computation. They outsourced road network data to the server and only retrieve useful information in query processing. Both of the solutions, nevertheless, are inefficient and not scalable since they require quadratic storage to the number of nodes and/or a large number of costly PIR operations.

In this paper, we focus on developing efficient and privacy-preserving solutions for shortest path queries. Specifically, we employ the concept of *oblivious storage* (OS) [4], which provides a general privacy model that not only guarantees data confidentiality but also conceal data access patterns. With the help of this model, we can process queries in a trusted environment with useful data privately retrieved from untrusted servers. However, to conceal access patterns, data stored in the server need to be shuffled and re-encrypted periodically. This leads to high amortized costs for I/Os. Thus, the performance of an OS application would be heavily influenced by I/O efficiency and storage capacity.

We design a new oblivious shuffle algorithm, *Interleave Buffer Shuffle* (IBS), to optimize the OS scheme proposed in [5]. This makes our data retrieval operations much faster than that of the PIR protocol used in [3]. We also proposed two schemes, *Compact Key point Index* (CKI) and *Shortcut Passage Index* (SPI), for shortest path queries. By leveraging shortcuts generated by *Arterial Hierarchy* (AH) [6] and adopting modified KD-tree as partitioning strategy, both schemes provide much better performance and scalability than the state of the art [3]. We also propose several novel optimizations on data organizing and indexing.

In summary, our contributions lie mainly on the following aspects:

- We design Interleave Buffer Shuffle (IBS) to optimize the OS scheme proposed in [5].
- We develop specific schemes for shortest path queries, which achieve much better performance and scalability than the state of the art [3] with a strong privacy property in practical scenarios.
- We enhance our schemes with novel optimizations on data organizing and indexing.

- We conduct comprehensive experiments on real-world road networks and assess trade offs between different schemes.

The remainder of this paper is organized as following. We first survey related works and techniques in Section II. Next, we give a formal definition of the problem and describe our system model in Section III. In Section IV, we present an OS scheme with a new oblivious shuffle algorithm. Then, we describe our scheme for private shortest path queries in Section V and VI. Furthermore, we make discussion on providing better privacy properties and several trade-offs for OS scheme in Section VII. We show our experimental results in Section VIII and conclude the paper in Section IX.

II. Related Works

In this section, we do a brief survey on classical shortest path problem and existing solutions for private shortest path computation based on different methodologies.

Shortest Path Queries on Road Networks: The typical solution for finding shortest paths, Dijkstra's algorithm [7], is often inefficient on large road networks. To address this problem, plenty of techniques have been proposed. Some of them [8]–[11] pay more attention to the practical performance, which can answer most shortest distance queries within several milliseconds on a million-nodes road network [10]. However, their performance against long distance queries may be not good enough. Other solutions [12]–[15] focusing on theoretical complexity are worst-case efficient, but they often require a high preprocessing time and space overheads (e.g. Proposed solution in [12] consumes about 1 GB space for a road network with 10^5 nodes) .

Arterial Hierarchy (AH) [6] bridges theory and practice together. On the theoretical side, AH answers any query in $\tilde{O}(k + log\beta)$ time, where k is the number of nodes on the shortest path, $\beta = d_{max}/d_{min}$, and d_{max} (d_{min}) is the largest (smallest) distance between any two nodes in the road network. On the practical side, it performs as the state of the art in terms of query time with moderate space overhead.

Private Shortest Path Queries: The obfuscation method for shortest path queries [1] introduces *Obfuscator* to the system model, which is a trusted third-party mediator between clients and the LBS. In this model, the client sends her shortest path query from source s to destination t to the Obfuscator. The *Obfuscator* extends s and t with a number of fakes to obfuscation sets S and T. These two sets are sent to the LBS with all pairs from $S \times T$ treated as queries. After receiving candidate paths from the LBS, the *Obfuscator* figures out the real answer and sends it back to the client. According to [1], to improve the performance, decoy sources and destinations should be chosen close to s and t. Obviously, obfuscation method reveals information about queries since the LBS provider knows that the real source and destination is in a small set and has a rough idea of their positions. Besides, this method discloses strong clues about the length and composition of the shortest paths as all query pairs proceeded would have similar length and potentially share plenty of edges.

Data-oblivious graph algorithms [2] are proposed to solve such information leakage. These algorithms will execute the same sequence of instructions and generates indistinguishable data access patterns for any inputs of the same length. In [2], several data-oblivious graph algorithms, including that for single source shortest paths (SSSP), have been proposed. However, this algorithm needs the graph structure stored in an adjacent matrix, which directly causes problems on scalability. Moreover, computation cost of the data oblivious SSSP algorithm is at the scale of $O(|V|^2)$, which is also not suitable for large graphs.

Another approach to address this problem is the PIR-based solution [3]. PIR is a primitive operation for retrieving data from a server while the server can hardly learn which item is retrieved. PIR offers cryptographic privacy guarantees based on the reductions of problems that are computationally infeasible or theoretically impossible to solve. Traditional PIR protocols involve considerable computation and/or communication overheads on sizable datasets. Towards the practicality of PIR, [16] raises a hardware-aided PIR protocol by introducing an off-the-shelf hardware secured co-processor(SCP). With the help of this protocol, [3] constructs two schemes, CI and PI, for shortest path queries. However, their overheads on preprocessing and auxiliary space are also unacceptable for big road networks.

Recently, Samanthula et al. propose a privacy-preserving protocol for shortest path discovery [17] based on a Dijkstra-like algorithm. However, it requires several minutes to answer a shortest path query over a small graph of 5000 vertices. In addition, related to the private shortest path query problem, an approach to answer approximate shortest distance queries using encrypted distance oracles has been proposed in [18].

III. Problem Formulation and System Model

In this section, we formally define shortest path queries and our privacy objective. Moreover, we show an overview of our system model and its designing paradigm.

Shortest Path Query: Let $G = (V, E)$ be a weighted graph which represents a road network. Each node $v \in V$ is located in Euclidean space with coordinates, and each edge $e \in E$ is associated with a positive weight $w(e)$. A path $P(s, t)$ where $s, t \in V$ is a sequence of edges $e_1, e_2, \cdots, e_n$ starting with s and ending with t. The cost of a path is defined as the sum of edge weight on it. A shortest path query $SP(s, t)$ is asking for a path from s to t with the least cost.

System Model: We propose a general system model for privacy sensitive applications, which consists of three components as following:

- **Outsourcing Storage.** An outsourcing storage provides a general key-value storage service (e.g. Amazon S3), where users can store massive data and only retrieve useful data needed by applications.
- **OS Handler.** Generally, data are encrypted for confidentiality. However, information can still be leaked from access patterns. To obfuscate the access patterns, we employ the conception of *Oblivious Storage (OS)*. OS handler is an instance of OS mechanism, which provides an interface between the application and outsourcing storage that secures the access patterns from leaking information to the service provider of outsourcing storage.
- **Application.** The application part contains execution logic designed for answering queries issued by clients. It

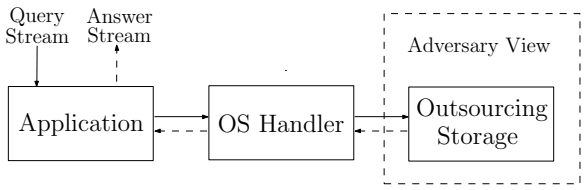


Fig. 1. Overview of the System Model

will run in a trusted environment. When receiving a query, the application will retrieve useful data from outsourcing storage through OS handler, compute the result and reply to the client.

Designing Paradigm: Although simple to apply such system model to any application, it is not free to use. Since OS will introduce non-trivial I/O overheads, we have to make applications I/O efficient so as to achieve a reasonable performance. In addition, the performance of OS Handler is also influenced by the total amount of outsourced data. This requires us to make outsourced data compact and wisely indexed. Another important observation is that the application should be light in computation since secured devices (personal computers, mobile phones or secure co-processors) only have limited computation power.

Adversary: The adversary in our model is the service provider of outsourcing storage with polynomial bounded computational power. We assume that it has identity information of clients and can extract access patterns for a specific application. It is free for the adversary to have knowledge of the whole privacy mechanism. The adversary is curious but not malicious, i.e., it wishes to gain information from clients' queries, yet it would always process requests correctly.

Privacy Objective: Our goal is to prevent the adversary from learning anything other than the length of access patterns from its view. Note that, as many other works [19], [20], we do not consider information leakage through the timing channel, such as when or how frequently the client makes requests. In the rest of this paper, we term this privacy objective as *Strong Query Privacy*.

Note that an OS mechanism guarantees that the adversary should be unable to distinguish any two possible access patterns of the same length. Along with the confidentiality provided by encryption, it is obvious that our system model satisfies strong query privacy. In addition, our work can be extended to achieve stronger privacy properties (to hide the length of access patterns as well), which will be discussed in Section VII.

IV. Oblivious Storage

In this section, we will introduce *Oblivious Storage* in details and explain how we choose appropriate OS scheme for our system model. Furthermore, we propose a new oblivious shuffle algorithm, *Interleave Buffer Shuffle* (IBS), to improve the performance and fix privacy issues of existing work [5].

A. OS Model

The conception of Oblivious Storage (OS) was first proposed in [4] as a practical approach of Oblivious RAM (ORAM) [22]. ORAM is a general model for the scenario when CPU has to access data stored in an untrusted RAM. Based on this original model, OS introduces application as a system component and relax the limits on client memory from $O(1)$ to a sub-linear scale. In recent years, as OS has stronger privacy objectives and more reasonable restrictions, term ORAM is also used to refer to oblivious storage as well.

Formally, in OS model, clients store their data outsourced and wish to keep the data private while running applications. Though traditional encryption schemes provide confidentiality, they cannot prevent information leakage through access patterns. To solve this, OS stores and fetches data in atomic units (referred as *blocks*). Note that we have to make all blocks the same size. Otherwise, the adversary can distinguish them easily by their sizes. In addition, each block is associated with an unique identifier. We pack a block and its identifier as an *item*. Throughout this paper, we use notation N to denote the total number of items that OS needs to support, which we refer as *capacity*.

From the perspective of architecture, OS model consists of a *Server* and a *Client*. *Server*, also referred as outsourcing storage in Figure 1, is a general *key-value* storage service providing following basic operations:

- $get(k)$: Return the value of the item with key k.
- $put(k, v)$: If the server contains an item with key k, then replace its value with v. Otherwise, add a new item (k, v) to the server.
- $getRange(k_1, k_2, p)$: Return the first p items in the server with keys in range $[k_1, k_2]$. If there are fewer than p such items, then all of them are returned.
- $delRange(k_1, k_2)$: Remove all items with keys in the range $[k_1, k_2]$ from the server.

Client, also referred as OS handler in Figure 1, holds a small amount of private memory where the client can perform computations protected from the server. In original OS model, client may pose two operations, $get(k)$ and $put(k, v)$, to retrieve or update items. In this paper, we only consider $get(k)$ operations that will never return $null$ (we call it a no-miss assumption). We do this since it is the only requirement for supporting our upper-level applications (i.e. shortest path queries). In fact, our OS scheme described later can be easily extended to a miss-tolerant version and support $put(k, v)$ operations with techniques introduced in [5].

Privacy. The main goal of OS is to hide access patterns. In this paper, we adopt the same privacy definition as [5], which consists of two aspects:

- **Confidentiality.** The adversary should not be able to know the contents of each item.
- **Hardness of Correlation.** The adversary should be unable to distinguish any two access patterns of the same length.

B. Choice of OS Schemes

In the past decades, numbers of OS schemes are proposed to protect user's access patterns. We roughly classify them into two categories based on their data lookup techniques:

Shuffle based OS schemes: [22] proposes two basic OS constructions: square-root solution and hierarchical solution. Both

TABLE I. COMPARISON OF PERFORMANCE

Model	Operation Performance		Storage		Failure Probability
	Online	Amortized	Client	Server	
IBS OS	$O(1)$	$O(\sqrt{N}/\alpha)$	$O(\alpha\sqrt{N})$	$N+\sqrt{N}$	No Failure
Practical OS [5]	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$N+\sqrt{N}$	$o(1)$
Usable PIR [16]	$O(\log_4 N * logN)$	$O(\log_4 N * logN)$	$O(\sqrt{N})$	$\geq 4N$	Negligible[1]
Path-ORAM[2][20]	$O(\frac{\log^2 N}{\log(Z/\log N)})$	$O(\frac{\log^2 N}{\log(Z/\log N)})$	$O(\log N \cdot Z)\cdot\omega(1)$	$16NZ$	$N^{-\omega(1)}$

[1] The probability of failure is bounded by $(\frac{e}{b})^b \frac{m}{1-e/b}$, where $b = \log N$ and $2\log N \leq m \leq \log^7 N$ [21].

[2] Z is the size of each block and $\omega(1)$ is a security parameter [20].

of them make use of server-end dummies and data shuffling to achieve privacy properties. Based on these constructions, plenty of schemes [4], [5], [16], [19], [23] have been proposed.

Index based OS schemes: In this category of schemes [20], an index structure is used for data lookup. Thus, it requires storing such index in client's memory. When the client memory cannot afford to store the index, it can be outsourced to the server in a similar way as data blocks at the cost of increased communication overhead.

Shuffle based OS schemes normally organize server storage in one or several layers. To protect user's privacy, they shuffle data blocks and dummies around the layers periodically. Schemes based on square root solutions [5], [22] take simple one-layer constructions, which helps them provide the lowest online latency and the least server-side storage consumption. However, they suffer from long system unavailable time caused by full data shuffles. To solve this problem, schemes based on hierarchical solutions [4], [19], [21] delay full data shuffles by introducing more layers and shuffle data between them. Nevertheless, complex storage structure and more frequent (partial) shuffles make these schemes need more server-end storage and get slower in online latency. Note that PIR protocol [16] adopted in [3] actually rests on a hierarchical solution based OS scheme.

The representative of index based OS schemes is Path-ORAM [20]. In Path-ORAM, the server-side storage is organized as a binary tree where each node contains a constant-size bucket for storing data. A data query will retrieve all blocks on the path which contains the query target, push them into client memory and then put another group of blocks back to the tree. Unlike shuffle based schemes, Path-ORAM have no system unavailable time as it does not involve data shuffles. Though its communication cost is acceptable in practice, overheads on server-side storage may pose as a big cost to the user.

Note that there are several works [24], [25] focus on OS schemes for secured multi-party computation (SMC), where several parties jointly perform a computation task on their private data such that all parties can obtain the computation result, but no party can learn the data from another party. Nevertheless, we employ OS under an outsourcing storage setting, thus such OS schemes are not suitable to our problem. Table I compares several OS schemes theoretically on query performance, storage consumption and failure probability. (IBS OS is the optimized OS scheme we introduce in this paper) We also conduct an experiment in Section VIII-A to show the performance of different schemes in practice.

As shown in Table I, IBS OS has the best online operation performance and the least server-side storage consumption. The main drawback is that it may cause a non-trivial system unavailable time periodically after numbers of operations. And it is only acceptable (within few seconds) when the capacity is not huge. In contrast, Path-ORAM does not involve any system unavailable time and has moderate online performance. Nevertheless, it requires much more server-side storage than other schemes. Note that the schemes to be introduced in V and VI for shortest path queries can work easily with any OS schemes. Thus, depending on the scale of the application, we make choices between IBS OS and Path-ORAM. If the size of outsourced data is moderate, IBS OS is the best choice. Otherwise, we pick Path-ORAM for its worst-case performance guarantee.

C. Square-Root Solution

Before we introduce our new data shuffle algorithm, we briefly describe the construction of a square-root solution. This OS construction framework requires the client has a memory size of $M = \sqrt{N}$ blocks at least. In fact, the client may holds more available memory. We can take this into consideration by using a notation α so that the size of client memory can be denoted as αM. (A typical value for α is 10 [5], [16], [26].)

Initially, the client encrypts and stores N original items and αM dummy items to the server. Note that we need to assign identifiers for each dummy item as well. We denote them as $-1, -2, \cdots, -\alpha M$ for convenience. For the sake of obliviousness, each item (i, b) is mapped to a substitute key using a pseudo-random hash function h_r, where r is a secret random nonce chosen by the client. Besides, each block b is encrypted as $En_{SK}(b)$, where En is a general encryption function with a secret key SK only known by the client. Thus, each item (i, b) will finally be stored as a new key-value pair $(h_r(i), En_{SK}(b))$ in the server.

When performing $get(i)$, the client first searches her own memory for an item with the identifier i. If she does not find one, she would request the item from the server by committing a request $get(h_r(i))$. Note that the server will always return an item $(h_r(i), En_{SK}(b))$ since we are under a no-miss assumption. Next, the client decrypts $En_{SK}(b)$ to get the block b and pushes the item (i, b) into the memory. On the other hand, if the client finds the item in her memory, she takes that item and issues a dummy request $get(h_r(-j))$ to the server, where j is a counter for next dummy key. In this case, when the item returns, the client would insert that dummy item into memory as well. Therefore, from the adversary view, the client is always requesting a distinct item with a random key.

After performing αM accesses, the client memory will be full. Then, the client blocks her requests temporarily and enters

a *rebuilding* phase. In the rebuilding phase, the client first cleans up its memory and sets counter j back to 1. Next, she shuffles the data in the server using her own private memory. During the data shuffle, the client will choose a new random nonce r' and another secret key SK' to encrypt each item (i, b) into a new version $(h_{r'}(i), En_{SK'}(b))$. As soon as the rebuilding phase is finished, the client can continue to access next αM items.

As the key technique to achieve *Hardness of Correlation*, data shuffle procedure should prevent the server from tracking any item. We can formulate this requirement as below:

Definition 1 (Oblivious Shuffle) If a procedure generates a uniformly random permutation of K items while the adversary is unable to track any item during the process, we call such procedure an oblivious shuffle.

Since oblivious shuffle will significantly affect the performance of OS schemes, large numbers of works [4], [5], [16], [27] have put efforts on designing efficient algorithms. [5] proposes *Buffer Shuffle Method* (BSM), which achieves linear computation (and communication) complexity. However, it may cause information leakage with a small probability, namely $o(1)$. To address this privacy issue, we design *Interleaved Buffer Shuffle*, which achieves a better performance as well.

D. Interleaved Buffer Shuffle

Interleaved Buffer Shuffle (IBS) is an oblivious shuffle algorithm requiring private memory of size $O(\sqrt{K})$. This algorithm can be stated in three stages:

Padding. To shuffle K items, we first insert several padding items to the original sequence so as to extend its length to the nearest square number T^2 (i.e. $T = \lceil \sqrt{K} \rceil$).

Permuting. Permuting stage will generate a random permutation of T^2 items coming from the padding stage, as shown in Algorithm 1.

In this algorithm, we first randomly permute each T items and map them to T different sets. After first round of permutation, each set would have T items from T different original groups. Then, we permute items in each set once more to get the final result.

We choose the modern version of Fisher-Yates shuffle [28] in step 5 and step 16, which can generate a random permutation of T items in $O(T)$ time. Besides, since the server can track the items by their contents, we also need to re-encrypt each item before putting it back to the server.

Lemma 1 *Procedure permuting() generates a uniformly random permutation of T^2 items for any integer T.*

Proof: For convenience, we define step 1 - step 11 of Algorithm 1 as round 1 and step 12 - step 22 as round 2, and let A_0 denotes the original order of T^2 items and A_1, A_2 denotes the order after round 1 and round 2 respectively.

To proof this lemma, we only need to show that $\forall 1 \leq i \leq T^2, 1 \leq j \leq T^2$, $Prob((A_0)_i = (A_2)_j) = 1/T^2$.

Algorithm 1: permuting()

```
1:  generate a new random nonce r' and secret key SK'.
2:  for i = 1 → T do
3:     retrieve T items by getRange().
4:     delete the returned items by delRange().
5:     generate a random permutation P of the T items.
6:     for j = 1 → T do
7:        re-encrypt item P_j with r' and SK'.
8:        bound a prefix "j :" to item P_j.
9:        put item P_j back to the server.
10:    end for
11: end for
12: generate a new random nonce r'' and secret key SK''.
13: for i = 1 → T do
14:    get items with prefix "i :" by getRange().
15:    delete the returned items by delRange().
16:    generate a random permutation P of the T items.
17:    for j = 1 → T do
18:       remove the prefix of item P_j.
19:       re-encrypt item P_j with r'' and SK''.
20:       put item P_j back to the server.
21:    end for
22: end for
```

For any item $(A_0)_i$ in A_0, after round 1, there are T possible locations across T different sets in A_1 where each location has a probability of $1/T$. Note that, during round 2, each set in A_1 is operated independently with each other. Namely, each possible location of $(A_0)_i$ in A_1 is permuted in another universe of T items, which results in $\forall 1 \leq i \leq T^2, 1 \leq j \leq T^2, Prob((A_0)_i = (A_2)_j) = 1/T \times 1/T = 1/T^2$. ∎

De-padding. After the permuting stage, the client asks the server to delete the padding items so as to get the final result. To show this is a random permutation of the original items, we introduce the following lemma.

Lemma 2 *Suppose permutation B_1 has $K+P$ items in which P items forms set D. If B_2 is a random permutation of B_1, $B_2 \backslash D$ is a random permutation of $B_1 \backslash D$.*

Proof: As B_2 is a random permutation of B_1, we have $Prob(B_2) = \frac{1}{(K+P)!}$. Taking out P items of set D, the new permutation $B_2 \backslash D$ has probability $Prob(B_2 \backslash D) = \frac{P! C_{K+P}^{K}}{(K+P)!} = \frac{1}{K!}$ to appear in all permutations of $B_1 \backslash D$. ∎

Theorem 1 *Interleaved Buffer Shuffle is an oblivious shuffle.*

Proof: According to Lemma 1 and Lemma 2, IBS generates a uniformly random permutation of K items for any integer K, which means the adversary cannot track an item by location. Combined with the fact that each item is unable to be identified by its content because of re-encryption, IBS satisfies Definition 1. ∎

Analysis on IBS. To shuffle K items with IBS, we need an additional server storage of size $O(\sqrt{K})$ and a client memory of size $O(\sqrt{K})$. As to the computation (and communication) cost, IBS is obvious linear to the number of the items.

Comparison with BSM. Buffer Shuffle Method (BSM) also

works in several rounds like IBS. In each round, BSM retrieves all data by groups, shuffles retrieved data within each group and puts them back to the sever, just as how IBS works. The difference is that BSM does not map data from one group to different groups for the next round. Such difference makes BSM cannot generate random permutations with equal probabilities. By the proof from [5], with probability $1-o(1)$ after 4 passes, one cannot guess the location of an initial key-value pair with probability greater than $1/K+o(1/K)$, where K is the number of items to be permuted.

Recall that IBS only needs to retrieve and put back each item 2 times whereas BSM needs 4 at least to bound fail probability within $o(1)$. Moreover, IBS provides a theoretical provable oblivious property. In contrast, although increasing round number can reduce the probability of information leakage, BMS cannot achieve perfect oblivious property in a limited number of rounds.

Analysis on OS Scheme. By applying IBS to the square-root solution, we now have a full implementation of our OS scheme (termed as IBS OS). In the following, we analyze this scheme from the aspects of space consumption, computation overhead and privacy guarantee.

Space Consumption: In order to construct a square-root solution, the server needs to hold at least $K = N + \alpha M$ items. Moreover, as required by IBS, we need to insert additional items to extend K to its nearest square number T^2. Mathematically, the maximum number of padding items used in IBS is $P_{max} = T^2 - (T-1)^2 - 1 = 2\lceil\sqrt{K}\rceil - 2$. Therefore, to build an oblivious storage of capacity N, our OS scheme requires $O(N + \sqrt{N})$ space in the server and $O(\sqrt{N})$ on the client side.

Computation Overhead: For each online request of OS, our scheme only needs to retrieve one item. Taking the rebuilding phase into consideration, amortized overhead for each request is $O(\sqrt{N}/\alpha)$ as the cost for IBS is $O(N)$.

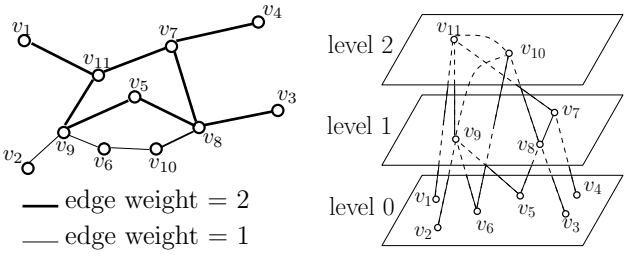
Note that the OS service would be unavailable when it is in the rebuilding phase, which may be not that trivial as N grows. To handle this, we can adopt techniques like replicas into our scheme to cut down such unavailable time. We will discuss this in Section VII.

Privacy Guarantee: With IBS, we can prove our scheme guarantees the privacy objectives required by the OS model.

Theorem 2 *Our OS scheme with Interleaved Buffer Shuffle is a privacy-preserving Oblivious Storage implementation.*

Proof: Apparently, *Confidentiality* is guaranteed by the encryption.

For *Hardness of Correlation*, suppose a sequence of data accesses $get(h_r(i_1))$, $get(h_r(i_2))$ ... $get(h_r(i_n))$ is observed. We just need to prove the probability that the adversary can figure out sequence $i_1, i_2 \cdots i_n$ is $1/K^n$. According to Theorem 1, the adversary is unable to track any item after a shuffle process. Together with the facts that all requests between two shuffles are distinct and items cannot be recognized by their encrypted contents, we can conclude that the adversary can only identify the requested item correctly with a probability of $1/K$. Hence, all possible access sequences of length n can be deduced by the adversary with the same probability $1/K^n$. ■



(a) Road network G. (b) Node hierarchy H.

Fig. 2. AH Overview [6]

V. Compact Key Point Index

With the provable privacy properties guaranteed by the OS model, we can now start designing schemes for shortest path computation based on the system model and its designing paradigm described in Section III.

[3] proposes two schemes, CI and PI, based on a hardware-aided PIR protocol. Since they share similar challenges in scheme designing with ours, a simple solution is to apply their schemes directly to our system model. Nevertheless, both of their schemes have problems in prohibitive communication or storage cost over large road networks. CI requires too many block retrievals for a single shortest path query, while PI involves huge storage overheads. Besides, both of them have very long preprocessing time. To address these problems, we develop two new schemes, CKI and SPI, which need fair numbers of I/Os and moderate storage overheads. Moreover, our preprocessing overhead is reasonable for large road networks.

In this section, we will describe our first scheme, *Compact Key point Index* (CKI), which aims at minimizing outsourced data size while achieving a reasonable low query latency. CKI leverages shortcuts generated by an index structure, *Arterial Hierarchy* (AH) [6], to reduce block retrievals. Road network data and its AH index are partitioned into data blocks and then outsourced to the server. While answering shortest path queries, CKI does Dijkstra-like traversals on the AH index with required data blocks retrieved through the OS handler.

A. Overview of AH

AH [6] is an index structure designed for answering shortest path queries over road networks. It performs as the state of the art in terms of query time with moderate space overheads and preprocessing time. AH introduces the concept of *arterial edge* to select important edges and nodes from the road network. By applying this concept, AH assigns a *level* to each node, where nodes with higher level are of more importance. Based on these levels, AH generates shortcuts to speed up query processing. Specifically, a shortcut $\langle u,v\rangle$ is created if the shortest path from u to v only passes through nodes whose levels are lower than both u's and v's. In addition, each shortcut $\langle u,v\rangle$ is associated with a *key point* w, where w is on the shortest path from u to v and the length of $\langle u,v\rangle$ equals the sum lengths of $\langle u,w\rangle$ and $\langle w,v\rangle$ ($\langle u,w\rangle$ or $\langle w,v\rangle$ can be a shortcut or an edge).

For example, we have the road network G as shown in Figure 2(a). AH constructs a three-level hierarchy H as shown in Figure 2(b), where each level contains a disjoint subset of the nodes. H includes not only all edges, but also two shortcuts, $\langle v_9,v_{10}\rangle$ and $\langle v_{10},v_{11}\rangle$. Length of each shortcut equals to that of the original shortest path between two ends.

To find shortest paths on this index, we can perform two Dijkstra-like traversals starting at the source and the destination. During the traversal, we could always avoid traveling from a higher-level node to a lower-level node while maintaining the correctness of the algorithm. For instance, to answer query $SP(v_1, v_{10})$, we should start two traversals from v_1 and v_{10}. In particular, v_1 can only reach v_{10} and v_{11}, since (i) v_{11} is the only node adjacent to v_1, and (ii) we could only traverse to a non-lower-level node v_{10} from v_{11}. We get the shortest path as soon as two traversals meet. Recall that every shortcut $\langle u,v\rangle$ is associated with a key point w. Hence, the original shortest path in G can be constructed by recursively replacing each shortcut $\langle u,v\rangle$ by its corresponding two-hop path $\langle u,w\rangle$ and $\langle w,v\rangle$.

B. Data Partitioning

Though AH outperforms most alternatives in terms of query time, it is a memory-based index and does not care about how to format itself into fix-sized data blocks for I/O efficiency. As a result, we cannot apply AH directly to our system model. To build our scheme, we need to design partitioning strategies for road network data and its AH index.

Recalling the designing paradigm proposed in Section III, outsourced data should be compact and wisely indexed. This requires us making full use of each data block. More importantly, partitioning strategy should be friendly to query processing. In particular, it is better to put adjacent nodes in the same block to minimize the number of retrieved blocks. Last but not least, partitioning information (i.e. how we split the data) is expected to be concise since we want to keep them locally with the application. In CKI, we adopt a modified 2-dimensional KD-tree as our partitioning strategy. Nodes of the road network are clustered naturally with KD-tree. Furthermore, partitioning information only contains splitting coordinates which is tiny and easy to be kept locally.

A KD-tree partitions the whole space by spatial coordinates. Each KD-tree leaf is mapped to a spatial region and holds all nodes of the road network within it. Initially, there is only one leaf mapped to the whole space. Each time, we find a split line to split one leaf into two leaves of even size. To say the size of a leaf, we mean the total amount of essential information (including their coordinates, AH levels, adjacent edges, etc.) carried by the nodes in it. For example, in Figure 3, the KD-tree is constructed with split lines $x = 4$ (the first splitting coordinate), $y = 5$ and $y = 6$ (the second splitting coordinates in left and right children), etc. We keep splitting the leaves until the size of each leaf is no more than the block size (chosen by the user). If a leaf only contains one node, we stop splitting it as well. Finally, each leaf contains a set of nodes and is assigned to a data block. Note that, a node may carry large amount of information that cannot be fitted in a single block since we are also storing shortcuts. In that case, we simply assign two or more blocks to it.

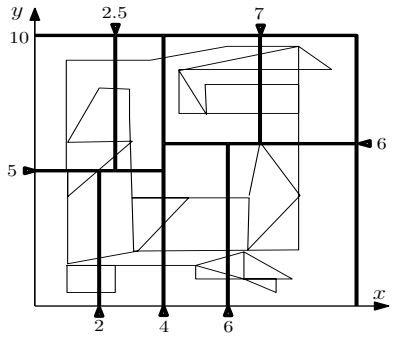


Fig. 3. KD-tree partitioning

KD-tree can organize road network data and its AH index into fixed-size blocks indeed, yet it also cause serious problems on low block space utilization. We observe from the experiment (in Section VIII-C) that simply adopting KD-Tree leaves about 20%-40% block space unused. To solve this problem, we modify the construction of KD-tree by changing splitting mechanism and merging small blocks together. This effectively increases the utility of data block space. Besides, level information generated by AH index also helps us to achieve better partitioning locality. We leave the details of these optimizations in Section V-D.

Recall that AH constructs the final answer recursively by replacing each shortcut with its corresponding two-hop path. If we put key points along with the shortcuts, we have to retrieve all blocks that contain nodes of the original shortest path. Unfortunately, most information carried by these blocks is useless. In addition, more information carried by a node potentially means less nodes in a single block. As a consequence, more blocks are required for a query, which would eventually pose a big impact on performance. Since the procedure of building original shortest path is completely independent of Dijkstra-like traversals, we pick all key points out and partition them with KD-tree separately to avoid retrieving useless information.

After partitioning, road network data and its AH index are separated into following parts:

Road Network Data (F_d). F_d contains road network data and all shortcuts generated by AH. Edges and shortcuts are clustered according to their source nodes. For each edge or shortcut $\langle u,v\rangle$, F_d stores (i) v and its AH level, (ii) edge weight and (iii) spatial coordinates of v. We employ modified KD-tree to partition this part of data to fixed-size blocks.

Key Point Data (F_k). For each shortcut $\langle u,v\rangle$, F_k stores its key point w and which block contains w in F_k. Key point data are partitioned with modified KD-tree as well so that they are clustered according to shortcuts' source node.

Header (F_h). Header holds the partitioning information which guides CKI to fetch required data from F_d and F_k. It also contains global information like the number of nodes and edges in the road network.

As to the location of these data, F_d and F_k are outsourced by the OS scheme, while F_h are kept locally with the application.

C. Query Processing

Given source s, destination t and their coordinates as a shortest path query, CKI first constructs a shortest path with both edges and shortcuts based on data retrieved from F_d. CKI first figures out which blocks in F_d contain s and t by looking up F_h. Then, we fetch found blocks through OS handler and start performing two Dijkstra-like traversals concurrently starting from s and t. During the traversal process, if any information cannot be found locally, the block containing it will be fetched by OS handler. Note that CKI would not go to a lower-level node due to the property of AH index, which greatly reduces the amount of nodes we need to try. Two traversals stop as soon as they meet and a shortest path (with shortcuts) is found.

The remaining part is to recursively replace shortcuts by their original shortest paths based on data from F_k. For each shortcut $\langle u,v\rangle$, CKI first gets which block containing u in F_k by looking up F_h with u's coordinates. Then, CKI fetches that block and get the key point w of the shortcut. Such block will also provide which block in F_k contains w. Thus, it can continue to replace $\langle u,w\rangle$ and $\langle w,v\rangle$ recursively.

D. Optimizations

In this section, we show our optimizations for reducing storage consumption and enhancing query performance.

Small blocks combination: As mentioned in Section V-B, we choose KD-tree as our partitioning technique. However, simply adopting KD-tree leads to low block space utilization (around 60%). The reason is that we need to assign exactly one block size (denoted by B) to each leaf. However, there might be leaves with size much smaller than B, leaving most block space unused.

To address this issue, an intuitive method is to merge several low utilized blocks together into one block. We propose a simple greedy algorithm to do this. Assume n blocks $b_1, ..., b_n$ are sorted by their information size in ascending order and the pointer p has an initial value n. Consider $b_1, ..., b_n$ in ascending order of their subscripts: for each b_i, we find the biggest $j \leq p$ such that the sum size of b_i and b_j does not exceed the block size B, then set $p = j$ and merge b_i into b_p. If we cannot find such j, no blocks can be combined and the algorithm terminates. If a leaf with size L cannot fit in one block ($L > B$), we simply regard it as a leaf of size $r = (L \mod B)$ and run the same algorithm.

Unbalanced partitioning: Normally, KD-tree only splits a leaf equally. Thus, most leaves are more than 50% utilized. However, only blocks with utilization under 50% can be optimized by small blocks combination. In order to enhance its effect, we construct a KD-tree with unbalanced leaves.

Suppose we have a leaf of size $S \in (B, 2B]$ and we want to find a split line to split it into two leaves. Without loss of generality, we assume this line is vertical and the road network nodes in that leaf $(a_1, ..., a_n)$ are sorted according to their horizontal coordinates. In a standard KD-tree, split is made at the biggest i where the summed size from a_1 to a_i is less than $S/2$. All nodes to the left of a_i (including a_i) form the left child and the rest form the right child. In unbalanced partitioning, we change the splitting point to the biggest i so that the summed size from a_1 to a_i is less than B. This will produce a leaf of size close to B and another close to $S - B$. With a higher probability, some blocks with utilization under 50% will be generated.

Unbalanced partitioning itself does not affect space consumption since the number of blocks remains unaltered. But it is powerful when cooperating with small block combination. Combining these two optimizations effectively reduces generated data size by nearly 20% as shown in our experiments (see Figure 6) .

Hierarchical partitioning: In addition, we propose an optimized partitioning mechanism for road network data F_d. The main idea rests on the following observation: during the traversal, we always avoid traveling from high levels to low levels. As a consequence, we do not need the information about low-level neighbors of a node. Therefore, instead of partitioning entire road network together, we partition nodes on each level separately and integrate them into F_d.

Hierarchical partitioning is powerful against long distance queries. Since processing such queries usually needs more exploration in high levels, this optimization can help us avoid retrieving useless low-level nodes in query processing. Experimental results (Figure 6) show that this optimization decreases the number of retrieved blocks and query latency for long distance queries by 20%.

VI. Shortcut Passage Index

CKI achieves a moderate space consumption according to our experiments (see Figure 5(a)). In addition, it can find a shortest path with shortcuts in very few steps, which makes it appealing for shortest distance queries. Nevertheless, CKI faces the problem that some shortcuts may correlate to nodes spanning over a huge amount of F_k blocks when recursively constructing the original shortest path. Fetching all these blocks poses a considerable impact on query latency. *Shortcut Passage Index* (SPI) is proposed to address this problem. With additional space consumptions, SPI is able to retrieve only one block in average for the reconstruction of a shortcut.

In SPI, data in F_d and its partitioning information in F_h remain unmodified, but F_k is substituted by the shortcut passage data F_p:

Shortcut Passage Data(F_p): SPI pre-computes original paths of all shortcuts and stores them in F_p. In other words, F_p keeps a list of nodes $a_1, a_2, ..., a_n$ for each shortcut $\langle u,v\rangle$ where path $(u, a_1, ..., a_n, v)$ is a shortest path from u to v in original road network. With the help of F_p, SPI can directly replace each shortcut by its original path rather than constructs the path recursively with key points.

It is obvious that KD-tree is no longer suitable for F_p since we do not need to keep locality when indexing these shortcuts. Moreover, low space utilization of KD-tree will result in a large space consumption. Thus, we assign each shortcut to a block and then run the greedy algorithm described in small blocks combination. By our experiments, such algorithm gives us a space utilization more than 95%, which is much higher than that of KD-tree.

Note that F_h grows into a considerable size because the greedy algorithm does not provide a concise partitioning

information. To solve this problem, we store the partitioning information of F_p inside F_d. Specifically, we insert an identifier i to each shortcut stored in F_d, where i identifies the F_p block where we can find the original path of that shortcut. Hence, the partitioning information of F_p can be omitted from F_h.

Obviously, OS capacity required in SPI is much larger than that of CKI. As a result, I/O cost to fetch a data block through OS handler becomes higher. Nevertheless, SPI has better performance in terms of query performance, especially on online query latency, for long distance queries.

VII. Discussion

Further improvement on privacy. In Section III, we defined our privacy objectives with regardless of access pattern length and information leakage through timing channel. When these assumptions do not hold, the adversary would be able to learn some information (e.g. result length) about the queries.

To prevent such information leakage, we can extend our scheme by padding all possible access patterns through OS handler to the same length. Specifically, we denote this length as a padding goal PG. If a query only requires cnt blocks ($cnt < PG$) during its process, we need to issue another $PG - cnt$ dummy block requests. Combining with this method, our schemes guarantee that each query would always ask for the same number of blocks from the view of adversaries.

Obviously, PG should be large enough to support every possible query. In Section VIII-E, we employ the most straightforward way to find one, which is to pre-compute shortest paths for all possible $\{source, destination\}$ pairs and set the maximum number of block retrievals as PG.

We can also fix this problem, to some extent, by randomly issuing several dummy block requests after query processing. Though such method cannot achieve perfect privacy as padding access patterns to a padding goal, it is more light-weighted and gives better performance.

As to the problem on timing channel, existing work [29] provides mechanisms for bounding OS timing channel leakage to a user-controllable limit.

Tradeoffs on OS Schemes. Recall that there is a rebuilding phase in the construction of IBS OS, which will cause system unavailable for a period of time. As the capacity of OS increases, this unavailable time will become non-trivial.

Many works [4], [19], [30] have conducted study on background shuffling and eviction in OS construction using techniques like piggy-backed eviction or choosing a proportional shuffle rate to data accesses. These works amortize shuffle overhead upon online request latencies. If we want to keep the dominate online performance of IBS OS, we can introduce several *replicas* dealing with requests in turn while others are in their rebuilding phases. This is the most direct method to cut off system unavailable time without sacrificing any performance on online latency, while the drawback is that it requires more resources on both server and client.

VIII. Experiment

In this section, we evaluate our proposed schemes in practice. All our experiments are conducted between two machines (server and client) connected to a 100Mbps LAN. The server is a 64-bit Windows workstation with an 8-core Intel E5-2643 processor and 64GB RAM. It hosts an instance of MongoDB as outsourcing storage service. The client, a PC with Intel Q8400 CPU and 4GB RAM, hosts the application and the OS handler. All preprocessing are done by the server.

TABLE II. Dataset Characteristics

Name	Corresponding Region	Number of Nodes	Number of Edges
SCA	California (small)	21048	43386
DE	Delaware	48812	119004
VT	Vermont	95671	209286
NA	North America	175813	358204
CO	Colorado	435666	1042400
FL	Florida	1070376	2687902
CA	California	1595557	3919120
TX	Texas	2037133	5061230

We adopt SHA-256 and AES/CFB from Crypto++ Library [31] as our hash function and encryption function respectively in all implemented OS schemes. We take 4KB as the block size, use a key length of 128 bits for AES encryption, and set α (defined in the square-root construction) to 10. Thanks to the authors of [3], we obtain the original implementation of CI and PI.

All datasets used in our experiments are shown in Table II. These datasets, each of which corresponds to a part of real road network, are available at [32], [33]. Following previous work [6], [12], we generate ten sets of shortest path queries $Q_0, Q_1, ..., Q_9$ for each dataset, where Q_i contains 1000 pairs of (s, t) as queries and the shortest distances for each query are between $2^{i-10} l_{max}$ and $2^{i-9} l_{max}$. (l_{max} is the maximum distance of any two nodes in the dataset.)

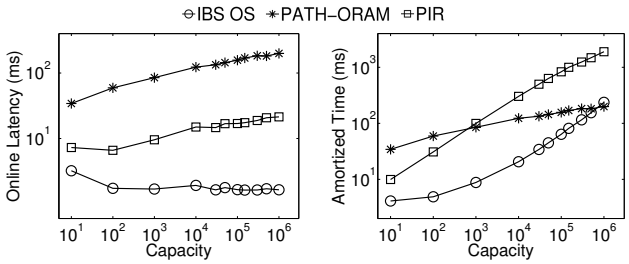
The remainder of this section is organized as following: We first conduct an experiment to examine our choice on OS schemes in Section VIII-A. Then, we evaluate storage and preprocessing overheads of both our schemes and the state of the art [3] in Section VIII-B. Next, we show the efficiency of our optimizations in Section VIII-C. In Section VIII-D, we show the query performance of CKI and SPI based on different OS schemes and give a comprehensive comparison between our schemes and the state of the art [3]. Finally, we evaluate the performance of our schemes after adopting padding goals to achieve further privacy properties in Section VIII-E.

A. Comparison of OS schemes

We implement three OS schemes: Path-ORAM, hierarchical OS based PIR [16] (referred as PIR for convenience below) and IBS OS as representatives for index, hierarchical and square-root based OS schemes. We evaluate them with the respect of two factors, online latency and amortized time, against OS capacity[1]. *Online latency* denotes the average query latency excluding system unavailable time, while *amortized time* takes the overhead of oblivious shuffle procedure into consideration.

As shown in Figure 4(a), online latency of PIR and Path-ORAM grow sub-linear to capacity, while that of IBS OS stays at a constant around 1.6 milliseconds. IBS OS achieves dominate online performance because it only needs to retrieve one item for online requests. As to Path-ORAM, since no

[1]Unit for OS capacity is the number of items.



(a) Online Latency (b) Amortized Time

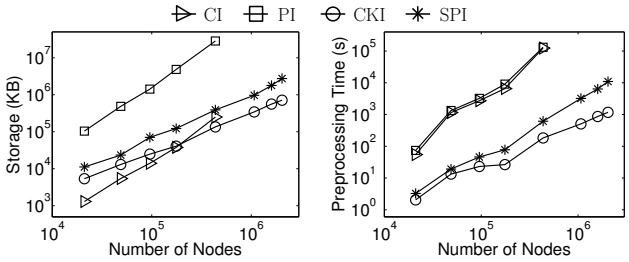
Fig. 4. Comparison of OS schemes

system unavailable time is involved, all its efforts for shuffling data are amortized to online latency. PIR introduces several layers and shuffle the data between them to delay full data shuffles. This requires PIR retrieving an item from each level, which slows down its online performance.

In Figure 4(b), we observe that IBS OS outperforms the other two in amortized time when capacity is below 10^6. This seems counter intuitive as the amortized complexity of IBS OS ($O(\sqrt{N})$) is apparently greater than those of Path-ORAM ($O(\log n)$) and PIR ($O(\log^2 n)$). The reason why we have this result is that IBS OS has a much lower constant factor thanks to the simple construction of square-root solution and the good performance of IBS.

B. Storage and Preprocessing

Figure 5(a) shows the summed storage size of all outsourced data. Note that PI is only available in the first 3 datasets since its storage overhead is terribly high, which increases quadratically with the size of road network. Particularly, the storage size of PI exceeds 27GB for dataset CO whose raw data size is only around 30MB. CI costs the least on small datasets, yet its storage size grows faster than that of our schemes. In contrast, the space consumption of CKI and SPI keeps low and increases linearly with $|V|$.



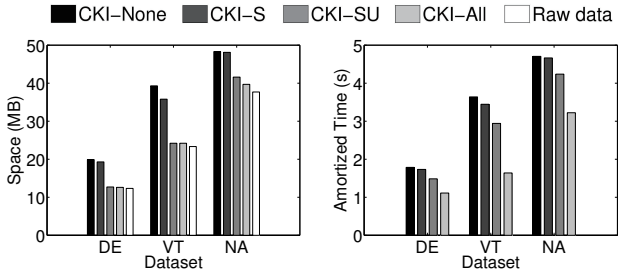
(a) Summed Storage Size (b) Pre-computation Costs

Fig. 5. Space and Pre-computation Costs

Figure 5(b) presents our results on preprocessing time . CKI's preprocessing time keeps the lowest on all dataset and exhibits a linear increase with $|V|$. SPI grows slightly faster due to a sorting procedure on all edges and shortcuts. On the other hand, CI's (also PI's) preprocessing time is super-linear to $|V|$, which limits its scalability on large road network. Specifically, CI costs more than a week to generate outsourced files for dataset CO.

C. Efficiency of Optimizations

As CKI and SPI share similar structures for F_d, optimizations make similar effects on them. Thus, we only show the results of CKI here. We implement four versions of CKI as below:



(a) Summed Storage Size (b) Amortized time

Fig. 6. Storage Size and Amortized Time

- *CKI-None*: CKI using plain KD-tree partitioning, i.e. without any optimization.
- *CKI-S*: CKI using the KD-tree with *small block combination*.
- *CKI-SU*: CKI using the KD-tree with *small block combination* and *unbalanced partitioning*.
- *CKI-All*: CKI with all optimizations, which is the version we use in the rest of other experiments.

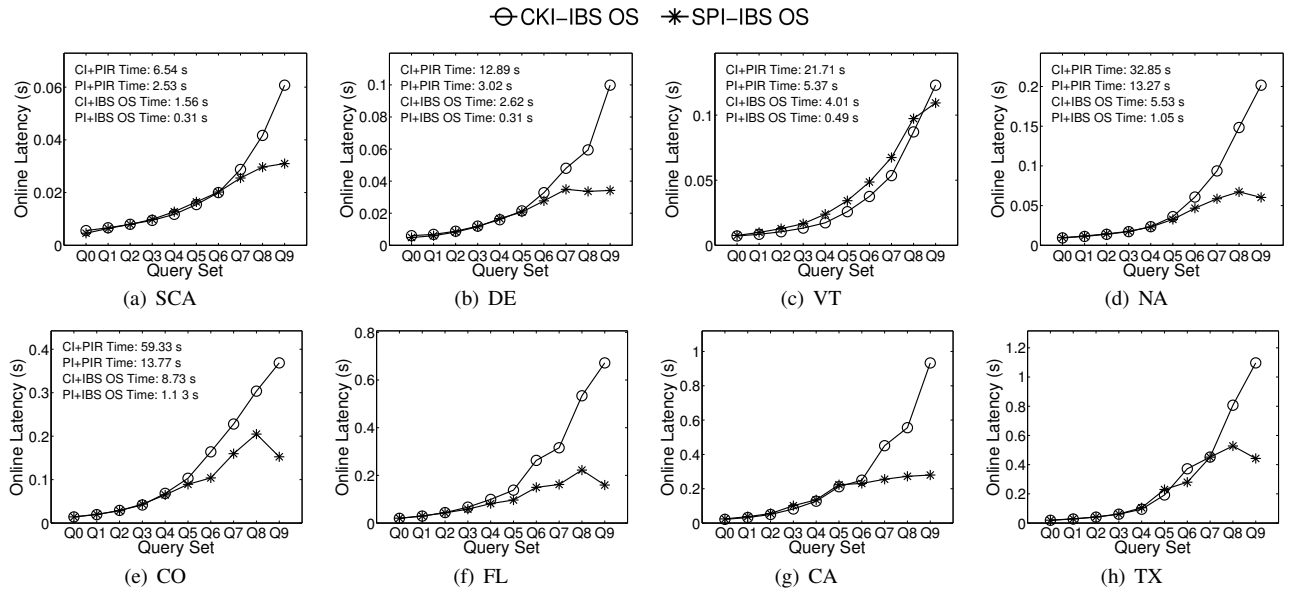
Figure 6(a) shows space consumption on datasets DE, VT and NA. For comparison, we also present the size of raw data (i.e. unformatted data). CKI-None constantly has the highest storage overhead with the lowest space utilization between 60% and 80%. As we mentioned in Section V-D, only using *small block combination* (CKI-S) does not help much, but it remarkably reduces storage overheads together with *unbalanced partitioning* (CKI-SU). Meanwhile, we notice that *hierarchical partitioning* also slightly decreases the storage overhead even though it is originally introduced to cut down query time.

To show how these optimizations influence query performance, we run queries from Q_9 for dataset DE, VT and NA. Though first two optimizations would not make any difference in query processing, reduction on OS capacity leads to better query performance. On the other hand, applying *hierarchical partitioning* greatly reduces query latency, which is consistent with our discussion in Section V-D.

D. Query Performance

We build these CKI and SPI based on both IBS OS and Path-ORAM. This gives us a better view of choices on OS schemes within a real application. In addition, to show the efficiency of our schemes against CI and PI, we build CI and PI based on their original PIR protocol and IBS OS. Recalling that Path-ORAM does not involve any system unavailable time, *its online latency equals exactly to its amortized time*. To show our results under a better axis scale, we only put the results for Path-ORAM in Figure 9.

As shown in Figure 7, SPI evidently outperforms the others in terms of online latency, and is especially efficient against long distance queries: SPI's online latency is lower than that of CKI by 25%-50% on query set Q_9. Furthermore, online latency of SPI increases little on query sets Q_7, Q_8, Q_9 for most datasets (sometimes even decreases), which indicates the efficiency of AH shortcuts and our passage index. In contrast,



(a) SCA (b) DE (c) VT (d) NA

(e) CO (f) FL (g) CA (h) TX

Fig. 7. Online Latency

although providing stronger privacy properties, CI and PI run much slower than our schemes on all datasets even when we replace PIR by IBS OS.

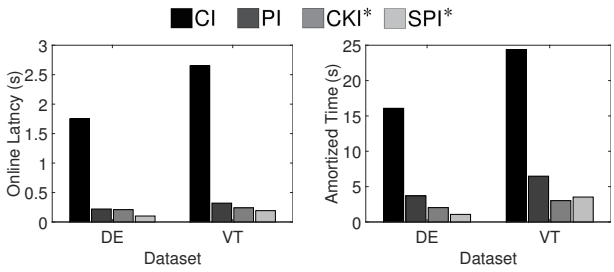
For amortized time in Figure 9, SPI is suppressed by CKI on most short distance query sets (e.g. Q_0 to Q_6). The reason is that SPI has a relatively large storage consumption, which leads to longer shuffle time. Nonetheless, SPI still performs well against long distance queries (Q_7, Q_8, Q_9), while CKI's amortized time grows fast on these query sets. Again, the response time of CI and PI is higher than that of CKI and SPI.

As to CKI and SPI based on Path-ORAM, we observe that the Path-ORAM version slightly outperform the IBS OS version in amortized time on large datasets (e.g. FL, CA, TX). We observe that Path-ORAM based schemes have non-trivial online latencies (10-30 seconds) even on moderate size datasets. Nevertheless, it does not involve any system unavailable time, which gives much better worst-case performance guarantee on large datasets. In contrast, IBS OS based schemes suffer from long system unavailable time on large datasets, which is around 6 minutes for CKI and 25 minutes for SPI on dataset TX.

E. Further Privacy Improvement

In Section VII, we mention that our schemes can achieve further privacy properties by padding all possible access patterns to the same length. The most straight-forward way is to find the maximum number of block retrievals. Specifically, in the preprocessing phase, we run our schemes for all source-destination pairs $\{s, t\}$ and record all $cnt_{s,t}$ which is the number of OS block retrievals in that query. Finally, we set the padding goal to $max\{cnt_{s,t} | \forall s, t \in V\}$. Processing all possible source-destination pairs is time-consuming, but we can easily speed up such procedure by paralleling it in a cluster.

We denote CKI and SPI with padding as CKI* and SPI*. Figure 8 shows their query performance on datasets DE and VT. Note that both online latency and amortized time of CKI* and SPI* are greater than those of CKI and SPI as shown in Section VIII-D, as padding access patterns involves dummy block retrievals. Even though, the performance of CKI* and SPI* is still better than CI and PI in all settings.



(a) Online Latency (b) Amortized time

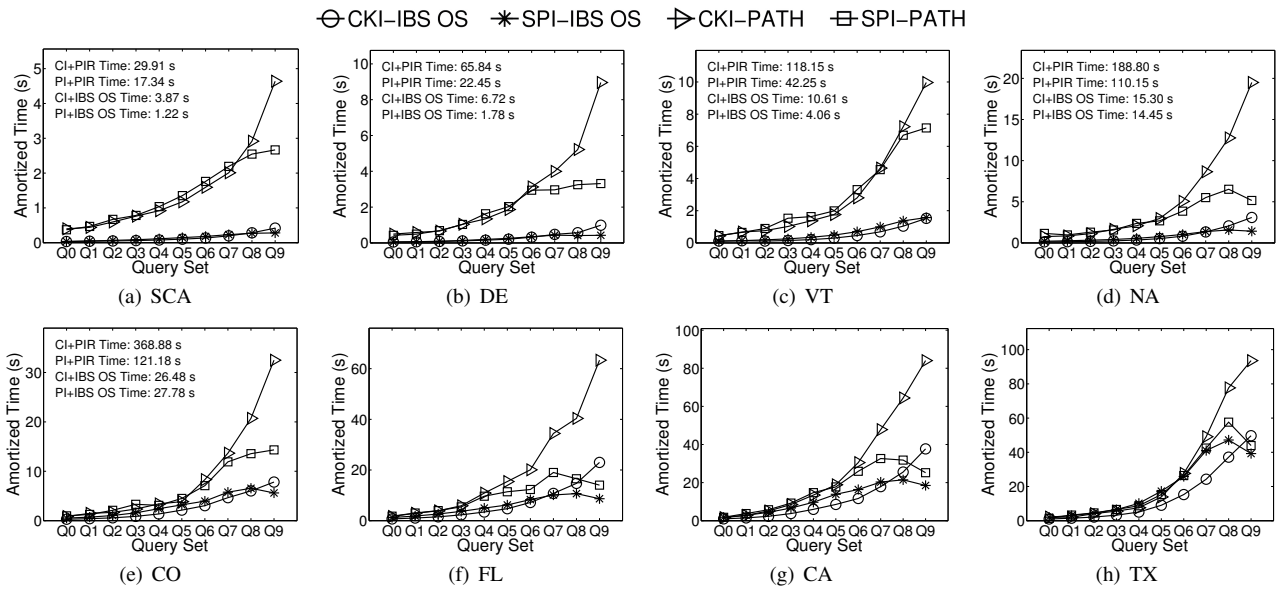
Fig. 8. Schemes with Improved Privacy

IX. Conclusions

In this paper, we introduce a general system model based on OS to support queries requiring strong privacy. To achieve better performance and privacy guarantee, we propose a novel oblivious shuffle algorithm and apply it on a general OS construction. Moreover, we design specific schemes for shortest path queries. Our schemes achieve good performance and scalability while still guaranteeing a strong privacy property. According to experimental results, our schemes perform well on real datasets and greatly advance previous work [3] on performance and scalability. In future works, we will focus on involving more privacy concerns and designing schemes for other location-dependent queries.

Acknowledgement

Bin Yao (corresponding author) is supported by the National Basic Research Program (973 Program, No.2015CB352403), the NSFC (No. 61202025, 61428204), the Scientific Innovation Act of STCSM(No.13511504200, 15JC1402400) and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939). Xiaokui Xiao was supported



(a) SCA (b) DE (c) VT (d) NA

(e) CO (f) FL (g) CA (h) TX

Fig. 9. Amortized Time

by grant ARC19/14 from MOE, Singapore and a gift from MSRA and AT&T, respectively.

REFERENCES

[1] K. C. Lee, W.-C. Lee, H. V. Leong, and B. Zheng, "Navigational path privacy protection: navigational path privacy protection," in *CIKM*, 2009.

[2] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *ACM Symposium on Information, Computer and Communications Security*, 2013.

[3] K. Mouratidis and M. L. Yiu, "Shortest path computation with no information leakage," *PVLDB*, 2012.

[4] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: making oblivious RAM practical," *Technical Report*, 2011.

[5] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Practical oblivious storage," in *ACM Conference on Data and Application Security and Privacy*, 2012.

[6] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *SIGMOD*, 2013.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, 1959.

[8] H. Bast, S. Funke, and D. Matijevic, "TRANSIT-ultrafast shortest-path queries with linear-time preprocessing," in *DIMACS Implementation Challenge*, 2006.

[9] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Science*, 2007.

[10] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: faster and simpler hierarchical routing in road networks," in *Experimental Algorithms*, 2008.

[11] A. V. Goldberg and C. Harrelson, "Computing the shortest path: a search meets graph theory," in *SODA*, 2005.

[12] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: an experimental evaluation," *PVLDB*, 2012.

[13] P. N. Klein, S. Mozes, and O. Weimann, "Shortest paths in directed planar graphs with negative lengths: a linear-space $o(nlog^2n)$-time algorithm," *ACM Transactions on Algorithms*, 2010.

[14] S. Mozes and C. Sommer, "Exact distance oracles for planar graphs," in *SODA*, 2012.

[15] J. Sankaranarayanan, H. Samet, and H. Alborzi, "Path oracles for spatial networks," *PVLDB*, 2009.

[16] P. Williams and R. Sion, "Usable PIR," in *Network and Distributed System Security Symposium*, 2008.

[17] B. K. Samanthula, F.-Y. Rao, E. Bertino, and X. Yi, "Privacy-preserving protocols for shortest path discovery over outsourced encrypted graph data," in *2015 IEEE International Conference on Information Reuse and Integration (IRI)*, 2015.

[18] X. Meng, S. Kamara, K. Nissim, and G. Kollios, "Grecs: Graph encryption for approximate shortest distance queries," in *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[19] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," *CoRR*, 2011.

[20] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path O-RAM: An extremely simple oblivious RAM protocol," in *ACM Conference on Computer and Communications Security*, 2013.

[21] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious ram and a new balancing scheme," in *ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[22] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, 1996.

[23] D. Q. Jinsheng Zhang, Wensheng Zhang, "S-oram: A segmentation-based oblivious ram," in *ACM Symposium on Information, Computer and Communications Security*, 2014.

[24] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "Scoram: Oblivious ram for secure computation," in *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[25] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[26] S. Papadopoulos, S. Bakiras, and D. Papadias, "Nearest neighbor search with strong location privacy," *PVLDB*, 2010.

[27] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *Automata, Languages and Programming*, 2011.

[28] R. Durstenfeld, "Algorithm 235: random permutation," *Communications of the ACM*, 1964.

[29] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *HPCA*, 2014.

[30] E. Stefanov and E. Shi, "ObliviStore: high performance oblivious cloud storage," in *IEEE Symposium on Security and Privacy*, 2013.

[31] Http://www.cryptopp.com/.

[32] Http://www.cs.utah.edu/ lifeifei/SpatialDataset.htm.

[33] Http://www.dis.uniroma1.it/challenge9/download.shtml.