Practical Access Pattern Privacy by Combining PIR and Oblivious Shuffle

Zhilin Zhang
Simon Fraser University & Amazon
zhilinz@sfu.ca

Ke Wang*
Simon Fraser University
wangk@cs.sfu.ca

Weipeng Lin
Simon Fraser University
weipeng_lin@sfu.ca

Ada Wai-Chee Fu
CUHK
adafu@cse.cuhk.edu.hk

Raymond Chi-Wing Wong
HKUST
raywong@cse.ust.hk

ABSTRACT

We consider the following secure data retrieval problem: a client outsources encrypted data blocks to a semi-trusted cloud server and later retrieves blocks without disclosing access patterns. Existing PIR and ORAM solutions suffer from serious performance bottlenecks in terms of communication or computation costs. To help eliminate this void, we introduce "access pattern unlinkability" that separates access pattern privacy into short-term privacy at individual query level and long-term privacy at query distribution level. This new security definition provides tunable trade-offs between privacy and query performance. We present an efficient construction, called SBR protocol, using PIR and Oblivious Shuffling to enable secure data retrieval while satisfying access pattern unlinkability. Both analytical and empirical analysis show that SBR exhibits flexibility and usability in practice.

KEYWORDS

access pattern unlinkability, PIR, oblivious shuffling

ACM Reference Format:
Zhilin Zhang, Ke Wang, Weipeng Lin, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2019. Practical Access Pattern Privacy by Combining PIR and Oblivious Shuffle. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3357975

1 INTRODUCTION

Data outsourcing allows a cloud provider (termed *server*) to take over complicated and expensive tasks of storing and managing data for cloud users (termed *client*). The servers are widely considered as "semi-trusted" or "honest-but-curious", in that they follow the protocol honestly but may passively attempt to learn protected information from all data observed during the execution of the protocol. For this reason, outsourced data are crucially encrypted by the client. However, *access patterns*, referring to when and how often outsourced data are accessed, can still disclose sensitive information of queried data, such as keyword information of encrypted emails [19], ordering information of encrypted values [20], and even accurate plaintext of encrypted database [15, 25]. In particular, these access pattern-based attacks typically work as follows. The cloud server knows query distribution as auxiliary knowledge. Due to access pattern disclosure, the server also knows which encrypted data is retrieved by each encrypted query. Over time the server learns access distribution of outsourced data and thus can infer query contents by correlating the query distribution to the data access distribution. Once the content of an encrypted query is known, the server easily infers certain sensitive information of outsourced data retrieved by this query.

Under the outsourcing scenario, hundreds of works have been presented to enable secure query processing over outsourced data in the cloud, see [4, 36] for example. Many of these methods serve the common purpose of privately *searching* query answers and informing the client of their locations on the server. Depending on the query intent, there are different mechanisms to identify the answer's locations for a user query, such as SQL queries [16], range query [18], nearest neighbor query [39, 41], etc. However, there is one more step to be taken before completing the query, that is, the client needs to privately *retrieve* the encrypted answers from the server using the location information obtained. During this step, protecting access pattern becomes a major concern because simply accessing the identified locations immediately discloses access patterns of the query.

To fill this critical gap between searching query answers and retrieving the answers, this work focuses on a practical and efficient retrieval of query answers from the cloud server while protecting access patterns, given the locations of query answers. We assume that the searching step is done without disclosing access patterns, which is the focus of most existing works and beyond the scope of this paper, and we focus on the retrieval step for fetching a requested data given its location in the cloud.

1.1 Secure Block Retrieval

We consider a database $\mathcal{D}$ of τ data blocks $B_1, \cdots, B_\tau$, each of size m. The block size m is measured by the number of encryption units for accommodating the data of a block. For example, assuming one unit allows 1Kb data for encryption, a data block B_i containing 1Mb data would have the block size $m = 1024$, represented as a column vector of length 1024 with one element per encryption unit. At

*Ke Wang was partially supported by a discovery grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

initialization, the client will encrypt each B_i to $[B_i]$, and outsource the encrypted database $[\mathcal{D}] = ([B_1], \cdots, [B_\tau])$ to the server.

We consider the retrieval step as the following ***secure block retrieval*** query: the client aims to retrieve a single data block B_i from the outsourced database $[\mathcal{D}]$ given its block identifier i (i.e., the location). Our solution is designed to satisfy the following goals.

- **Security goals**. *Data confidentiality*: the server should not learn any plaintext of $\mathcal{D}$. *Access pattern confidentiality*: the server should not learn the value of i, i.e., which block in $\mathcal{D}$ is retrieved by the query. Data confidentiality is usually provided through encrypting D to $[\mathcal{D}]$ by an encryption scheme with a strong security guarantee such as semantic security. We focus on access pattern confidentiality in the rest of the paper.
- **Performance goals**. Computing query answers and achieving security goals involve *communication cost*, *client computation cost*, and *server computation cost*. Practical secure block retrieval requires that all three parts of query cost are reasonable small.

Developing a practical access pattern protection mechanism for secure data retrieval is challenging. Most existing schemes don't protect access patterns for efficiency concerns [36]. For example, [16, 18, 39, 41] assume a static relationship between each block B_i and the location of $[B_i]$ in $[\mathcal{D}]$ and allow the server to identify the query answers, which immediately discloses access patterns. Current techniques for access pattern protection including Private Information Retrieval (PIR) [5, 12, 22, 40] and Oblivious RAM (ORAM) [10, 30] are widely considered theoretical but impractical [24, 30]. PIR replaces a single query request with a full set of requests for the entire database, which introduces unacceptable computation for each query. ORAM continuously shuffles the blocks in a manner oblivious to the server as they are accessed, which leads to heavy communication costs (due to downloading the blocks to the client for shuffling and uploading permuted blocks to the server). For example, the communication cost of searchable symmetric encryption schemes using ORAM could be larger than that of simply sending back all outsourced data stored in the server [8].

1.2 Contributions

Our contributions are summarized as follows:

Contribution 1 (Section 3). We present a new notion of access pattern confidentiality, called *access pattern unlinkability*, to address the performance bottleneck of existing techniques. This notion divides access pattern into short-term pattern (i.e., which block is retrieved by each query) and long-term pattern (i.e., how often a block is retrieved over time), and thus allows specifying different parameters to protect them according to both security and performance impacts.

Contribution 2 (Section 4). We present a construction of secure block retrieval that provides access pattern unlinkability, called the *SBR Protocol*, using PIR and Oblivious Shuffling as building blocks.

Contribution 3 (Section 5). We analyze the complexity of two versions of SBR designed for different application scenarios, and compare them with state-of-the-art competitors with an in-depth analysis.

Contribution 4 (Section 6). We show experimentally that SBR approaches outperform state-of-the-art methods.

2 PRELIMINARIES

Our SBR Protocol adopts *Private Information Retrieval* (PIR) and *Oblivious Shuffling* (OS) as building blocks. In this section, we review existing PIR and OS schemes and explain how we choose appropriate schemes for our solutions.

2.1 Private Information Retrieval

Private Information Retrieval (PIR) [5, 12, 22, 37, 40] is a cryptographic primitive which allows a user to retrieve an item from a server in possession of a set of items without revealing which one is retrieved. Since PIR can hide the target of each retrieval, it is widely used independently or as a building block for hiding access patterns. In this paper, we consider only single-server, plaintext-independent PIR schemes. Multi-server schemes require multiple cloud providers and plaintext-dependent schemes [5, 22] require the data being in plaintext. Both are not appropriate for our setting.

2.1.1 Linear PIR. Wu et al. [37] have shown that an efficient single-server, plaintext-independent PIR can be built using the generalized Paillier cryptosystem ε_s ($s \geq 1$) [9]. ε_s is an additive homomorphic encryption scheme providing semantic security. Let N be the public key of ε_s. For any integer $s \geq 1$, the encryption algorithm can map any plaintext $x \in \mathbb{Z}_{N^s}$ to a ciphertext $[\![x]\!]$, where $\mathbb{Z}_{N^s} = \{0, 1, \cdots, N^s - 1\}$. By exploiting additive homomorphic property of ε_s, the following homomorphic dot product is developed in [37]:

$$\begin{aligned} [\![\vec{x}]\!] \odot \vec{y} &\overset{\text{def}}{=} [\![x_1]\!]^{y_1} \times \cdots \times [\![x_l]\!]^{y_l} \mod N^{s+1} \\ &= [\![\vec{x} \cdot \vec{y}]\!] \end{aligned}$$

where $x_i, y_i \in \mathbb{Z}_{N^s}$, $\vec{x} = (x_1, \cdots, x_l)$, $\vec{y} = (y_1, \cdots, y_l)^T$.

Using homomorphic dot product, for any $1 \leq i \leq l$, [37] implements PIR functionality of privately retrieving the i-th item of $\vec{y}$ on the server without disclosing i to the server as follows:

1) **GenerateQuery**: the client builds a binary selector $\vec{x} = (x_1, \cdots, x_l)$ with $x_i = 1$ and $x_j = 0$ for all $j \neq i$, encrypts $\vec{x}$ bit-wise to $[\![\vec{x}]\!]$ with ε_s, and sends $[\![\vec{x}]\!]$ to the server.
2) **QueryProcess**: the server computes the homomorphic dot product between $[\![\vec{x}]\!]$ and $\vec{y}$, which outputs the query result $[\![y_i]\!]$. The server returns the result $[\![y_i]\!]$.
3) **DecryptResult**: the client decrypts $[\![y_i]\!]$ to obtain y_i.

The client can adopt the above PIR protocol for retrieving the i-th data block from a column of l encrypted outsourced blocks $[C] = ([B_1], \cdots, [B_l])^T$ without disclosing i to the server, that is, by letting $\vec{y} = [C]$, we have $[\![\vec{x}]\!] \odot \vec{y} = [\![\vec{x} \cdot ([B_1], \cdots, [B_l])^T]\!] = [\![[B_i]]\!]$. Note that we use different notations for the encrypted binary selector $[\![\vec{x}]\!]$ and the outsourced data blocks $[C]$ because they are encrypted using different schemes. Any existing encryption scheme can be used for $[C]$; the only requirement is that the ciphertext space of $[C]$ is in the plaintext space of ε_s (i.e., $\mathbb{Z}_{N^s}$), which can be achieved by choosing s properly (more details will be given in Section 6). We wrap this function as the following primitive and use it directly in the remainder of this paper.

$$B_i \leftarrow PIR(i, [C]) \quad (1)$$

Table 1: Comparison of best oblivious shuffling schemes over n data blocks of size m

Best OS Algorithms		Communication cost	Client computation cost	Server computation cost
Client-side shuffling	Interleave Buffer Shuffle [38]	$O(mn)$	$O(mn)$	–
Server-side shuffling	Repeatable Oblivious Shuffle [42]	$O(n^2)$	$O(n^2)$	$O(mn^2)$

Due to the security properties of PIR, the client can use the primitive in Eqn (1) to privately retrieve any block B_i from $[C]$ without disclosing its location i and the content to the server.

Considering that $[C]$ contains l data blocks of size m, the cost of PIR operation in Eqn (1) includes: $O(l)$ and $O(m)$ client computation for preparing the query $[\![\vec{x}]\!]$ and decrypting the result $[\![[B_i]]\!]$, $O(l)$ and $O(m)$ communication for uploading the query $[\![\vec{x}]\!]$ and downloading the answer $[\![[B_i]]\!]$, and $O(ml)$ server computation for computing homomorpic dot product between $[\![x]\!]$ and $[C]$. Therefore, each call of this PIR operation incurs $O(m + l)$ client cost, $O(m + l)$ communication cost, and $O(ml)$ server cost.

2.2 Oblivious Shuffling

Oblivious shuffling is a cryptographic primitive which moves outsourced data blocks around in the server's storage in a fashion that disallows the server to correlate the previous physical locations of the blocks with their new locations. In other words, oblivious shuffling enables the client to permute a row of n outsourced data blocks $[\mathcal{R}] = ([B_1], \cdots, [B_n])$ in the server according to a permutation π chosen by the client without disclosing π to the server. We wrap this function as the following primitive and use it directly in the remainder of this paper.

$$[\mathcal{R} \cdot \pi] \leftarrow OS(\pi, [\mathcal{R}]) \tag{2}$$

Existing oblivious shuffling falls into two general categories.

Client-side shuffling depends on data movements to perform shuffle operations. To be specific, existing schemes in this category [13, 14, 26, 28, 38] commonly work in a multi-round manner. In each round, the client downloads a small portion of outsourced data to its local storage, shuffles it after decryption, re-encrypts the data and writes it back to the server. The best current practice of client-side shuffling is Interleave Buffer Shuffle [38].

Server-side shuffling leverages server computation to perform shuffle operations. Layered Shuffle [1, 10] achieves server-side shuffling through computing homomorphic matrix multiplication between outsourced data and encrypted permutation matrix on the server. However, it adds one more encryption layer to the outsourced data after every shuffle. To avoid unbounded increase of shuffling cost (as well as data size) due to layer explosion, Layered Shuffle still depends on data movements to periodically, say after every k shuffles, download outsourced data for peeling off extra encryption layers.

Repeatable Oblivious Shuffle [42] is recently proposed to completely eliminate the movement of outsourced data between the client and the server. This scheme essentially depends on the server to compute homomorphic dot products between outsourced data and some helper instructions that encode a target permutation, without increasing the encryption layer. Repeatable Oblivious Shuffle is currently the most efficient server-side shuffling scheme.

Table 1 summarizes the comparisons between the best client-side shuffling method (Interleave Buffer Shuffle [38]) and the best server-side shuffling method (Repeatable Oblivious Shuffle [42]) for shuffling n data blocks of size m. Interleave Buffer Shuffle involves no server computation, but its client cost and communication cost depend on both the block size m and the block number n, i.e., $O(mn)$, which is only acceptable when the block size m is small. In contrast, Repeatable Oblivious Shuffle eliminates the effect of m on the communication and client computation costs, but incurs a quadratic cost in n. Therefore, it suits the situation in which the size m of outsourced data block is large in relation to the number of blocks n.

3 ACCESS PATTERN UNLINKABILITY

To eliminate the drawbacks of existing techniques, we explore possible directions of access pattern protection and propose a new access pattern privacy definition that well balances the security and performance requirements in this section.

Considering block retrieval queries in Section 1.1, access patterns refer to the client's behaviors of accessing outsourced data blocks. To hide access patterns, a straightforward way is to hide the target block among all outsourced blocks whenever a retrieval query is requested. However, this strategy incurs prohibitive query cost. For example, PIR suffers from heavy server computation and ORAM suffers from heavy communication, as discussed in Section 1.1. To alleviate this performance bottleneck, one strategy is to hide the target block among only a group of outsourced blocks for each query. This, however, leads to uneven accesses to different outsourced blocks and opens up possibilities for attacks. To examine the security implication of these different strategies for access pattern protection, we separate access patterns at individual query level and at query distribution level: the **short-term pattern** tells which data block is retrieved by a particular query, and the **long-term pattern** shows how frequently a particular block has been retrieved (i.e., access distribution of the blocks over time).

Due to short-term pattern disclosures, the server may infer the information of an outsourced block, by correlating a known query with a subsequent observation of retrieving this block. To prevent such attacks, we simply hide the target of each query among a group of *indistinguishable* blocks. That is, if any block B_i is requested, a group of outsourced blocks containing $[B_i]$ are accessed in an indistinguishable way. Let the size of the group be some integer $l > 1$. The server would have only $\frac{1}{l}$ probability to infer which accessed block is actually retrieved by the query. Since enforcing such strict protection is expensive and it is widely assumed that the server is unable to acquire known queries [21, 34, 41], the group size l can be sufficiently small for promoting practical performance.

Due to long-term pattern disclosures, the server can launch more realistic attacks such as [15, 19, 20, 25] for inferring sensitive information. Recalling that these attacks are commonly executed by mapping the query distribution known as background knowledge to the access distribution over outsourced data. The latter is observed by the server. The successful linking critically relies on the server's observing a skewed (actual) access distribution. To prevent such attacks, it is sufficient to break the mapping by ensuring that observed accesses to outsourced blocks follow a uniform distribution from the server's perspective.

It is notable that enforcing the uniform observed access distribution over the whole database for hiding long-term pattern is unnecessary and outrageously expensive, especially when the total number of blocks is huge. For each outsourced block, assuming the server observes a uniform access distribution over a group of r encrypted blocks (including itself), then the server's probability of mapping this block to a known query distribution through its observed access frequency would be bounded by $\frac{1}{r}$ and the probability of mapping all these r blocks correctly would be bounded by $\frac{1}{r!}$. By choosing a sufficiently large r, the inference attacks using long-term patterns (e.g., [15, 19, 20, 25]) can be effectively avoided with improved performance. Moreover, different from short-term pattern protection, enforcing a uniform observed access distribution over a group of r blocks does not require accessing the entire group for each query. This provides an opportunity to amortize the cost of protecting long-term pattern among multiple queries for better performance (more details will be given in Section 4).

The next definition formalizes our ideas of practical access pattern protection discussed above.

Definition 3.1 ((l, r)-Access Pattern Unlinkability). Let $r > l > 1$. We say that a block B_i has an unlinkable access pattern w.r.t. (l, r) if the following conditions hold:

1. Short-term privacy: If B_i is retrieved by a query, a group of l blocks containing B_i are accessed indistinguishably.
2. Long-term privacy: At any time, B_i belongs to a group of r blocks whose observed access distribution $\mathscr{O}$ is a random sample of uniform distribution $\mathscr{U}$, noted as $\mathscr{O} = \mathscr{U}$.

Our security goal is that every block B_i in the database has an unlinkable access pattern. □

Intuitively, short-term privacy bounds *strictly* the probability of breaching short-term pattern to be $\leq \frac{1}{l}$, and long-term privacy bounds *statistically* that of breaching long-term pattern to be $\leq \frac{1}{r}$. This idea of bounding the inference probability by hiding the target among a group of candidates is shared by well known privacy measures such as k-anonymity [35] and l-diversity [23]. We stress that our definition makes no assumption about the query distribution; even if query distribution is highly skewed, it cannot be linked to the observed access distribution over outsourced data because the latter is a random sample of uniform distribution and every block could be a candidate.

We believe that limiting the risk of short-term and long-term pattern disclosure with two separate parameters l and r would provide a more practical solution to reconcile the security and performance goals. The key consideration in choosing l and r is the trade-off between security and efficiency: a larger l immediately increases each query's cost for better short-term privacy, whereas a larger r leads to a higher cost for better long-term privacy that, however, can be amortized over multiple queries. Motivated by the observation that long-term privacy outweighs short-term privacy, performance and privacy can be trade-offed by choosing a small l and a large r.

It is interesting to note that PIR and ORAM correspond to the extreme case of setting l to τ, the total number of blocks in the database; in this case, short-term privacy implies long-term privacy (i.e., $r = l$) because the target of every query is indistinguishable from all other blocks in the outsourced database. Since PIR and ORAM do not differentiate privacy concerns on short-term and long-term patterns, but hide them with the same level of protection, the amortized cost is equal to the worst-case cost. Unfortunately, the requirement that hiding the target of each query (i.e., short-term pattern) among a large group of data blocks is often impractical, especially for the group as large as the whole database.

4 OUR APPROACH

We now present our solution, called ***SBR*** to the secure block retrieval problem in Section 1.1, which achieves access pattern unlinkability in Definition 3.1.

4.1 Main Ideas

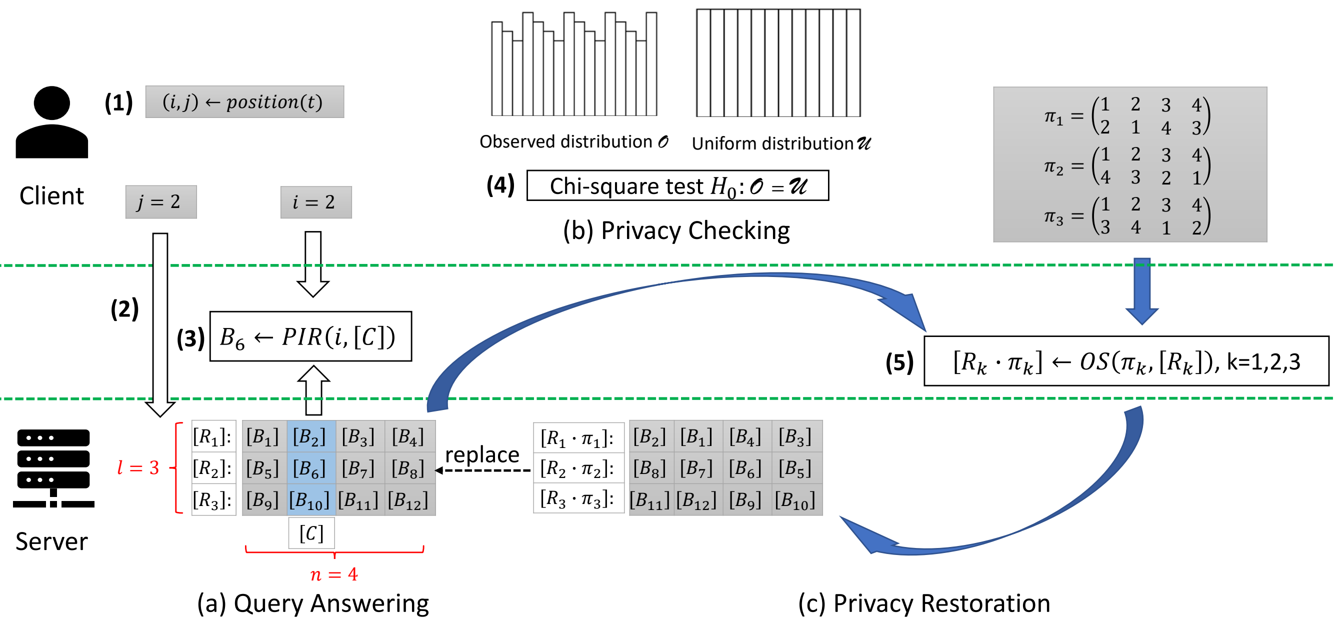
Given the security parameters (l, r), for simplicity we assume that r is divisible by l and let $n = \frac{r}{l}$. The outsourced database $[\mathcal{D}]$ is partitioned into *buckets* of r encrypted blocks each. Since each bucket is considered independently, our discussion below focuses on a single bucket. We organize the r encrypted blocks in a bucket as an $l \times n$ matrix. For any query aiming at a target block $[B_t]$ in this bucket, SBR involves the following three components to retrieving B_t, while enforcing the access pattern unlinkability in Definition 3.1 w.r.t. the parameters (l, r).

1) **Query Answering**: This component retrieves the target block B_t from the server while providing short-term privacy. To retrieve B_t, the client first determines the position (i, j) of $[B_t]$ within the bucket of r data blocks stored in the server, and sends j to the server. The server extracts the j-th column of the bucket currently containing $[B_t]$, say $[C]$. Since the i-th block of $[C]$ is the target block $[B_t]$, the client can adopt the PIR primitive described in Eqn (1) to privately retrieve B_t without disclosing i to the server, i.e.,

$$B_t \leftarrow PIR(i, [C])$$

During this operation, the server learns j for extracting the column $[C]$. However, the server does not learn i due to the property of PIR that the client privately retrieves B_t from $[C]$ without disclosing its position i in $[C]$. Therefore, short-term privacy is provided.

2) **Privacy Checking**: Through the query answering, the server learns that some block in column j was retrieved by the query. The client must evaluate if long-term privacy still holds (Definition 3.1) in the presence of this disclosure. This can be done by the chi-square test for goodness of fit [11, 17]. Let $\mathscr{O}$ =



Client
(1)
$(i,j) \leftarrow position(t)$
$j = 2$
$i = 2$
(2)
(3) $B_6 \leftarrow PIR(i,[C])$
Observed distribution $\mathcal{O}$
Uniform distribution $\mathcal{U}$
(4) Chi-square test $H_0: \mathcal{O} = \mathcal{U}$
(b) Privacy Checking
$\pi_1 = \begin{pmatrix}1&2&3&4\\2&1&4&3\end{pmatrix}$
$\pi_2 = \begin{pmatrix}1&2&3&4\\4&3&2&1\end{pmatrix}$
$\pi_3 = \begin{pmatrix}1&2&3&4\\3&4&1&2\end{pmatrix}$
(5) $[R_k \cdot \pi_k] \leftarrow OS(\pi_k,[R_k])$, k=1,2,3
Server
$l = 3$
$[R_1]$: $[B_1]$ $[B_2]$ $[B_3]$ $[B_4]$
$[R_2]$: $[B_5]$ $[B_6]$ $[B_7]$ $[B_8]$
$[R_3]$: $[B_9]$ $[B_{10}]$ $[B_{11}]$ $[B_{12}]$
$[C]$
$n = 4$
replace
$[R_1 \cdot \pi_1]$: $[B_2]$ $[B_1]$ $[B_4]$ $[B_3]$
$[R_2 \cdot \pi_2]$: $[B_8]$ $[B_7]$ $[B_6]$ $[B_5]$
$[R_3 \cdot \pi_3]$: $[B_{11}]$ $[B_{12}]$ $[B_9]$ $[B_{10}]$
(a) Query Answering
(c) Privacy Restoration

Figure 1: SBR ($r = 12$ and $l = 3$): (a) the client uses PIR to privately retrieve the 2nd (i=2) block (B_6) from the 2nd (j=2) column $[C]$ of the bucket; (b) the client performs chi-square test to check if current observed access distribution $\mathcal{O}$ remains uniform; (c) the client uses oblivious shuffling to privately permute all $l = 3$ rows of the bucket independently.

$\{\mathcal{O}(1), \cdots, \mathcal{O}(r)\}$ be the observed accesses to the r blocks in the bucket with $\sum_{k=1}^{r} \mathcal{O}(k) = o$, and $\mathcal{U} = \{\mathcal{U}(1), \cdots, \mathcal{U}(r)\}$ be the uniform distribution with $\mathcal{U}(k) = \frac{o}{r}$ for all $1 \leq k \leq r$. The *null hypothesis* H_0 states that $\mathcal{O}$ and $\mathcal{U}$ come from the same underlying distribution, or equivalently, $\mathcal{O}$ is a random sample of $\mathcal{U}$, denoted by

$$H_0 : \mathcal{O} = \mathcal{U}.$$

The chi-square statistic χ^2 is defined by

$$\chi^2 = \sum\nolimits_{k=1}^{r} \frac{(\mathcal{O}(k) - \mathcal{U}(k))^2}{\mathcal{U}(k)}$$

Let p_v be the p-value of the above χ^2 statistic. According to the literatures [11, 17], the chi-square test fails to reject H_0 at the *significance level* α if

$$p_v \geq \alpha. \quad (3)$$

Otherwise, it rejects H_0 at this significance level, which means that the two distributions are different. In this case, we consider that long-term privacy is violated. The condition for applying chi-square test is that the average accesses per block is at least 5, i.e., $o \geq 5r$. The "warm-up period" refers to the period before this condition is met. During this period, we consider that long-term privacy holds trivially as the number of observations is too small to meaningfully reject H_0.

The above chi-square test involves two types of errors: (*Type I error*) the null hypothesis H_0 is rejected when H_0 is true; (*Type II error*) H_0 fails to be rejected when H_0 is false. These two error types are inextricably linked such that a reduction in one error increases the other. The choice of α depends on the focus of Type I or Type II error. Given the significance level α, there is a α chance of making Type I error if H_0 is rejected. To limit Type I error, a small significance level (e.g., $\alpha = 0.05$) is commonly used in the literature. In our context, we want to limit Type II error that corresponds to missing the detection of long-term privacy violation. In order to do this, as suggested in [11, 17], the significance level should be a large value so that the null hypothesis can be rejected quite easily. In other words, the null hypothesis is accepted only in those circumstances where the data conforms very closely with the claim of H_0. So we choose a large α, e.g., $\alpha = 0.95$.

3) **Privacy Restoration**: If H_0 is rejected during the above test, this component is activated to restore long-term privacy. In particular, for each row $[\mathcal{R}_k]$ of the bucket in the server, $1 \leq k \leq l$, the client randomly picks a permutation π_k and adopts the oblivious shuffling primitive in Eqn (2) to privately permute $[\mathcal{R}_k]$ according to π_k while keeping π_k secret from the server, i.e.,

$$[\mathcal{R}_k \cdot \pi_k] \leftarrow OS(\pi_k, [\mathcal{R}_k]), 1 \leq k \leq l$$

After this operation, the block $[B_t]$ at position (i, j) is moved to position $(i, \pi_i(j))$ after privacy restoration, therefore, to retrieve $[B_t]$ subsequently, the client would request the block at position $(i, \pi_i(j))$ in the bucket. To support these operations, the client can store a *position map*, such that $(i, j) \leftarrow position(t)$ means that block B_t currently locates at (i, j) of the bucket. After the shuffling, the client updates *position* for all r blocks of the bucket according to π_k, $1 \leq k \leq l$. The space for storing position map is linear to the number of blocks in the bucket, but is independent of block size. Since position map is commonly used in existing access pattern protection techniques, we omit describing it in detail here and refer readers to the literature [10, 33].

Long-term privacy is provided because any violation of long-term privacy, tested by rejecting H_0, leads to oblivious shuffling of every row $[\mathcal{R}_k]$ of the bucket and the shuffling resets the access distribution of blocks in the bucket because the access information cannot be accumulated due to the hidden correspondence of the positions before and after the shuffling. Before the restoration being triggered, the long-term privacy is trivially guaranteed because the server cannot distinguish observed access distribution from uniform distribution as long as H_0 is still accepted.

We use the next example to explain how SBR retrieves a block from a bucket of outsourced blocks while enforcing access pattern unlinkability.

Example 4.1. Consider the bucket of 12 outsourced blocks $[B_1]$, $\cdots$, $[B_{12}]$ represented by a 3×4 matrix in Figure 1, where $l = 3$ and $r = 12$. The upper part represents information known by the client and the lower part represents information kept by the server. The following steps are involved to retrieve the block B_6.

Query Answering (Figure 1(a)): The client first finds the current position $(2, 2)$ of the query target B_6 in the outsourced bucket (step 1) and sends the column position $j = 2$ to the server (step 2). Next, the server extracts $[C]$ as the 2nd column of the bucket, i.e., $[C] = ([B_2], [B_6], [B_{10}])^T$. Then, the client launches a PIR operation to retrieve the 2nd block B_6 of $[C]$ from the server (step 3). Since PIR hides the row position $i = 2$ of B_6 among $l = 3$ candidates, the short-term privacy is guaranteed.

Privacy Checking (Figure 1(b)): Through the operations above, the server learns that query result must be one of $l = 3$ blocks in $[C]$. To evaluate implication of such leakage for long-term privacy, After the above query, the client updates the observed access distribution $\mathscr{O}$ and tests if the null hypothesis $H_0 : \mathscr{O} = \mathscr{U}$ still holds w.r.t the uniform distribution $\mathscr{U}$ (step 4). If the chi-square test rejects H_0, a violation of long-term privacy is detected.

Privacy Restoration (Figure 1(c)): If there is a privacy violation detected by privacy checking, for each row $[\mathcal{R}_k]$ of the bucket, the client launches an oblivious shuffling for the $n = 4$ blocks in $[\mathcal{R}_k]$ according to a random permutation π_k, $1 \le k \le 3$ (step 5). For instance, the 1st row $([B_1], [B_2], [B_3], [B_4])$ is changed into $([B_2], [B_1], [B_4], [B_3])$. Since all $l = 3$ rows are randomly permuted and all permutations are hidden from the server, the observed access information $\mathscr{O}$ becomes invalid. Thus, the long-term privacy is restored. The new contents of the bucket is illustrated in Figure 1(c). □

4.2 Algorithm

Algorithm 1 summarizes the SBR protocol for retrieving a target block B_t, $1 \le t \le r$. The protocol is executed jointly by the client and the server. The server maintains the bucket of r outsourced data blocks as an $l \times n$ matrix, for $n = \frac{r}{l}$. The client maintains the position map *position* to track current locations of all r blocks in the bucket, and maintains the observed access distribution $\mathscr{O}$ of these blocks. Initially, $\mathscr{O}$ is all-zeros. The steps are as follows.

Algorithm 1 $B_t \leftarrow SBR(t)$

```
Require: The client has $\mathscr{O}$ and position; the server has $[\mathcal{R}_k]$, $1 \le k \le l$
  1: Client: $(i, j) \leftarrow position(t)$; send $j$ to the server
  2: Server: $[C] \leftarrow j$-th column of the bucket
  3: Client and Server: $\boxed{B_t \leftarrow PIR(i, [C])}$
  4: Client: update $\mathscr{O}$ and perform the chi-square test
  5: if the chi-square test rejects $H_0$ then
     (a) for $1 \le k \le l$ do
             - Client: randomly choose $\pi_k$
             - Client and Server: $\boxed{[\mathcal{R}_k \cdot \pi_k] \leftarrow OS(\pi_k, [\mathcal{R}_k])}$
     (b) Client: $\mathscr{O} \leftarrow \vec{0}$; update position
```

Line 1: The client finds the position (i, j) of B_t in the bucket using the position map, and sends the column position j to the server.

Line 2: The server extracts $[C]$ as the j-th column of the bucket.

Line 3: The client and server work collaboratively to retrieve the i-th block of $[C]$, i.e., B_t, using PIR without disclosing the value i to the server.

Line 4: The client updates the observed access distribution $\mathscr{O}$ of the bucket by increasing the access of all blocks in $[C]$ by 1. If the warm-up period is over, the client performs the chi-square test using $\mathscr{O}$.

Line 5: If the chi-square test rejects H_0, the client and the server collaboratively shuffle each row $[\mathcal{R}_k]$ of the bucket using oblivious shuffling. The client resets $\mathscr{O}$ and update the position map *position*.

4.3 Discussion

The promising performance of SBR is contributed by several factors. First, our access pattern unlinkability applies to each individual bucket of r blocks, instead of the entire database (containing τ blocks), which significantly reduces the overhead for enforcing access pattern protection. The idea of partitioning a large database into multiple buckets and employing access pattern protection over individual buckets for gaining better performance was previously suggested in [3, 31, 32], where the bucket size $r = \sqrt{\tau}$ was suggested. The other contributing factor is the division of short-term privacy and long-term privacy so that we pay only a small overhead for short-term privacy associated with each query, and reduce the overhead for a stronger long-term privacy by amortizing this overhead among multiple queries. We demonstrate these performance gains analytically in Section 5 and empirically in Section 6.

5 ANALYTICAL EVALUATION

This section analytically evaluates the performance of the SBR protocol presented in Section 4, and reports the analytical comparison of SBR with state-of-the-art competitors.

5.1 Complexity Analysis

For each query, the SBR protocol in Algorithm 1 is applied to the bucket of r outsourced blocks that contains its result. As shown in Figure 1, the bucket is organized as an $l \times n$ matrix, with $n = \frac{r}{l}$. We measure SBR's query cost by

$$\textit{query cost} = \textit{retrieval cost} + \frac{1}{n_q} \times \textit{shuffling cost} \quad (4)$$

n_q is the number of queries answered between two consecutive privacy restorations over the bucket, so the second term represents the *amortized shuffling cost* per query.

The *retrieval cost* is incurred by applying a PIR primitive to the column $[C]$ of l blocks (line 3, Algorithm 1). In this paper, we adopt Linear PIR [37] for SBR. Thus, the retrieval cost includes $O(m + l)$ client cost, $O(m + l)$ communication cost, and $O(ml)$ server cost (refer to Section 2.1 for cost analysis of Linear PIR).

The *shuffling cost* is incurred by applying an oblivious shuffling method to each of the l rows in the bucket (line 5(a), Algorithm 1). As discussed in Section 2.2, Interleave Buffer Shuffle [38] and Repeatable Oblivious Shuffle [42] are the best client-side/server-side shuffling scheme, respectively. Since SBR can work with both of them, we develop two versions of SBR in this paper: we call the SBR implementation adopting Interleave Buffer Shuffle as "SBR-IBS" and the implementation adopting Repeatable Oblivious Shuffle as "SBR-ROS". Table 1 reports the cost of two schemes for shuffling one row. Thus, the total cost of SBR-IBS and SBR-ROS for shuffling l rows is l times more. That is, SBR-IBS incurs $O(lmn)$ client cost and $O(lmn)$ communication cost, while SBR-ROS incurs $O(ln^2)$ client cost, $O(ln^2)$ communication cost, and $O(lmn^2)$ server cost. This shuffling cost is amortized over the n_q queries.

The complete query cost of each version of SBR is given by the sum of retrieval cost and corresponding amortized shuffling cost. We summarize query cost of SBR-IBS and SBR-ROS in Table 2. We can observe that SBR-ROS outperforms SBR-IBS (i.e., less communication cost) if the size m of outsourced data block is large in relation to the number of blocks n; otherwise, SBR-IBS is a better choice. Thus, depending on specific application scenarios (small-sized blocks vs. large-sized blocks), we make choices between SBR-IBS and SBR-ROS. For both SBR, a large n_q is effective for diminishing the effect of amortized shuffling cost on the total query cost. In this case, the actual query cost of SBR is approximated by its retrieval cost. We will study n_q empirically in the next section.

5.2 Analytical Comparisons

As mentioned previously, there are traditionally two ways to hide a user's access pattern: PIR and ORAM. We compare SBR with state-of-the-art PIR and ORAM for showing its advantages. Note that SBR involves only one bucket to answer a query. To ensure fair comparison, we develop the baselines by partitioning the outsourced database into buckets as SBR and applying PIR and ORAM to each bucket (instead of the entire database). Typically, we have the following baselines that enable secure block retrieval:

- The PIR baseline is built by applying a single-server plaintext independent PIR method to the bucket containing the target block of a query, which achieves *(l,r)-access pattern unlinkability* with $l = r$. Because PIR guarantees that the retrieval of any block within the bucket (of r blocks) is indistinguishable from the retrievals of other blocks.

  Since the current best bound of single-server plaintext independent PIR is achieved by *Linear PIR* [37], we adopt it in this baseline. That is, we treat the entire bucket as a column vector of size r and input it to Eqn (1). Therefore, the query cost is the cost of Linear PIR over r blocks, i.e., $O(m + r)$ client cost, $O(m + r)$ communication cost, and $O(mr)$ server cost.
- The ORAM baseline is built by applying an ORAM method to the bucket containing query target, which also achieves (l, r)-*access pattern unlinkability* with $l = r$ as ORAM completely conceals the actual access to each block within the bucket. We consider *Path ORAM* [33] in this baseline as it has the best performance among all ORAM constructions (according to [7]).

  Path ORAM [33] organizes the server storage as a binary tree of nodes containing Z blocks each, with $\log r$ levels. Z is a constant, i.e., 4. For each query, the client retrieves all of $O(\log r)$ blocks along the path containing the target, permutes and re-encrypts these blocks, and writes them back to replace the original path on the server. It incurs $O(m \log r)$ client cost and $O(m \log r)$ communication cost per query.

Table 2: Comparison of query cost (a bucket has r blocks of size m, $r > l$ and $n = \frac{r}{l}$)

Protocol	Communication cost	Client cost	Server cost
SBR-IBS	$O(m + l + \frac{lmn}{n_q})$	$O(m + l + \frac{lmn}{n_q})$	0
SBR-ROS	$O(m + l + \frac{ln^2}{n_q})$	$O(m + l + \frac{ln^2}{n_q})$	$O(ml + \frac{lmn^2}{n_q})$
PIR	$O(m + r)$	$O(m + r)$	$O(mr)$
Path ORAM	$O(m \log r)$	$O(m \log r)$	0

Table 2 summarizes the comparison of SBR and the baseline methods. From the table, we can observe that both baselines suffer from serious drawbacks in query performance. PIR has low communication cost by downloading only the target block, but incurs heavy server computation ($O(mr)$) due to computing over the entire bucket for each query. Path ORAM avoids server computation at the cost of heavy communication ($O(m \log r)$) since each query involves downloading $\log r$ blocks to the client and uploading another $\log r$ blocks to the server. In contrast, our SBR approach balances the goals of reducing communication and computation costs. First, SBR incurs limited retrieval cost. SBR takes advantage of PIR's low communication cost for query answering, while at the same time it limits the application of PIR over one column of l blocks (instead of the entire bucket) to reduce server computation. Second, the separation of short-term and long-term pattern protection provides SBR the opportunities to amortize shuffling costs over a large number of queries (i.e., large n_q). Thus, it effectively reduces the contribution of amortized shuffling cost in the total query cost (in Table 2, large n_q makes the terms having n_q trivial). The joint effects of limited retrieval cost and amortized shuffling cost enable SBR to achieve practical query performance. We will evaluate this advantage empirically in the next section.

Table 3: Parameters in the experiments

	Range	Default setting
Block number τ		2^{20}
Privacy parameter l	2^3, 2^4 2^4, 2^6, 2^7	2^5
Privacy parameter r	2^8, 2^9, 2^{10}, 2^{11}, 2^{12}	2^{10}
Law parameter δ	0.6, 0.8, 1.0, 1.2, 1.4	1.0
Block size m (MB)	10^{-3}, 10^{-2}, 10^{-1}, 1, 10	1

6 EXPERIMENTAL EVALUATION

This section reports our empirical evaluation of four methods discussed in Section 5. All methods are implemented in C++ with OpenMP parallel programming on a server machine 96 Intel Core i7-3770 CPUs at 3.40 GHz, and a client machine with 2 Intel Core e7-4860 CPUs at 2.60 GHz. Both run a Linux system.

The empirical comparisons are based on partitioning the outsourced database into buckets (each has r encrypted data blocks) and then applying different secure retrieval methods to each bucket for retrieving the blocks within this bucket, which enforces (l, r)-access pattern unlinkability (Definition 3.1). The implementation details of each method are summarized as below:

- *SBR.* Two SBR methods adopt different oblivious shuffling algorithms, which depend on different schemes to encrypt outsourced data and thus have varied ciphertext spaces. To enable query answering with the PIR primitive in Eqn (1) (i.e., Linear PIR [37]), the ciphertext space of outsourced data must be in the plaintext space of Linear PIR, which is achieved by choosing the proper parameter s for Linear PIR.

  SBR-IBS adopts Interleave Buffer Shuffle [38]. We follow the same setting as [38] to implement IBS, which encrypts data blocks with AES-128 and produces 128-bit ciphertext space. To implement Linear PIR, we choose 1024-bit key size by following [37] and let $s = 1$. The resulted 1024-bit plaintext space of Linear PIR is sufficient to hold the 128-bit ciphertext space of outsourced data.

  SBR-ROS adopts Repeatable Oblivious Shuffle [42]. We follow the same setting as [42] to implement ROS, which encrypts data blocks with Paillier Cryptosystem [27] using 1024-bit key size and produces 2048-bit ciphertext space. For Linear PIR, we choose 1024-bit key size as above but let $s = 2$. It produces 2048-bit plaintext space for Linear PIR to hold the 2048-bit ciphertext space of outsourced data.

- *PIR.* As the PIR baseline is built from the straightforward application of Linear PIR. It can be implemented by following the same settings as SBR-IBS. That is, the data blocks are encrypted by AES-128, while Linear PIR is implemented by choosing 1024-bit key size and $s = 1$.

- *Path ORAM* [33]. We adopt the open source library SEAL-ORAM [6] for the Path ORAM baseline, which encrypts outsourced data with AES-128.

Table 4: n_q of SBR w.r.t. privacy parameter l

l	2^3	2^4	2^5	2^6	2^7
n_q	368	769	1609	3155	10630

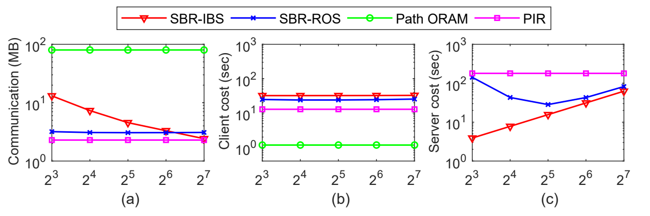


Figure 2: Query cost w.r.t. privacy parameter l (Path ORAM has no server computation and thus not reported in (c))

6.1 Experiment Setup

In the experiments, we first evaluate query performance of SBR with respect to its privacy parameters, i.e., short-term privacy parameter l and long-term privacy parameter r. Next, according to Eqn (4), SBR's query cost is also affected by n_q, the number of queries processed between two consecutive privacy restorations. n_q largely depends on the underlying query distribution. We simulate query distribution by the power law distribution, as commonly assumed in literature [2, 29]. That is, the frequency of retrieving the i-th block B_i in the database $\{B_1, \cdots, B_\tau\}$ is proportional to $1/i^\delta$. We evaluate the effect of varied δ on the query cost. Finally, since both data transmission and query computation are affected by the block size m, we also evaluate the effect of varied m on query cost[1].

The default values for some key parameters are as follows. We set the number τ of data blocks in the database to 2^{20} (the same as earlier studies in [24]). We set the long-term privacy parameter, r, to $\sqrt{\tau} = 2^{10}$ as explained in Section 4. We set δ, the skewness parameter of query distribution, to 1.0 (the same with the setting in [2] and [7]). Finally, we set the block size m to 1.0 MB as [24].

Table 3 summarizes the ranges of these key parameters and their default settings used in the experiments. For each specified setting of (l, r, δ, m), we generate a sequence of ten million queries randomly sampled from the underlying query distribution and report the average query cost. The reported n_q counts only the queries after the warm-up period of chi-square test; therefore, the actual count is larger than the value reported here. The significance level α for chi-square test is fixed at 0.95; we observe little difference in SBR's query cost for varying α from 0.80 to 0.99.

6.2 Experiment Results

1) Effect of privacy parameter l: A larger l offers stronger short-term privacy. Figure 2 reports the query cost w.r.t. l, while all other parameters are fixed at default. As l increases, the retrieval cost of SBR increases (because of more data blocks involved in query

[1]As different retrieval methods adopt different encryption schemes that vary the size of encryption units (e.g., one unit of SBR-IBS has 128-bit data but one unit of SBR-ROS has 1024-bit data), the block size m in the experiments is the size of a plaintext block in MB, instead of the number of encryption units in one block.

Table 5: n_q of SBR w.r.t. privacy parameter r

r	2^8	2^9	2^{10}	2^{11}	2^{12}
n_q	2459	1866	1609	1524	1437

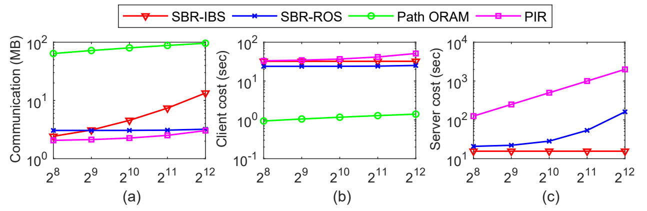


Figure 3: Query cost w.r.t. privacy parameter r (Path ORAM has no server computation and thus not reported in (c))

Table 6: n_q of SBR w.r.t. law parameter δ

δ	0.6	0.8	1.0	1.2	1.4
n_q	328553	16127	1609	406	166

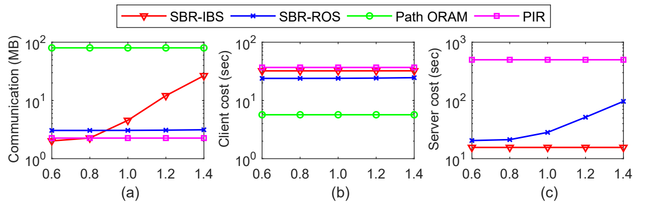


Figure 4: Query cost w.r.t. law parameter δ (Path ORAM has no server computation and thus not reported in (c))

answering) but the amortized shuffling cost decreases (mainly because of larger n_q). To balance these two components of query cost, a moderate l is recommended for optimal query performance. For example, with increasing l, SBR-IBS incurs reduced communication but enlarged server computation, while SBR-ROS exhibits first a decrease in query cost and then an increase (Figure 2 (c)). Both achieve better (or more balanced) query performance with $l = \sqrt{r} = 2^5$.

From Figure 2, we can observe that our SBR approaches significantly outperform the baselines with balanced communication and computation costs. To retrieve one block of 1 MB data, Path ORAM involves 80 MB data transmission and PIR involves approximately 200 seconds server computation. Thus, both are impractical for real applications. In contrast, with $l = 2^5$, SBR-IBS transmits only 4 MB data and computes the result within 15 seconds on the server. Similarly, SBR-ROS transmits 3 MB data and takes 28 seconds for computation by the server. As for the client's computation, the costs of all four methods are reasonably small (less than 30 seconds). These costs are mostly contributed by the decryption of query results. Note that Path ORAM has the highest client computation complexity ($O(m \log r)$ as shown in Table 2) but involves less time for computation. The reason is that Path ORAM does not require any server computation and thus adopts AES-128 as the encryption scheme, which enables fast decryption. In general, there results demonstrate the advantages of SBR. That is, SBR enables practical secure block retrieval by eliminating bottlenecks of the baselines (either prohibitive communication or server computation).

Considering that large n_q is essential for SPR's good query performance, we report the exact n_q with varied l in Table 4. Clearly, larger l leads to larger n_q. Such increased n_q is resulted from much smoother observed access distribution for two reasons. First, increased l achieves more balanced query accesses among different columns of the bucket. Second, the PIR-based query processing ensures that the accesses to more blocks within the same column are indistinguishable.

2) Effect of privacy parameter r: A larger r provides stronger long-term privacy but increases the row size $n = \frac{r}{l}$, thus, increases the shuffling cost of SBR. Figure 5 reports the query cost w.r.t. r, while other parameters are fixed at default. As r increases, the communication and client costs of Path ORAM increase, contributed by the $O(\log r)$ blowup in their cost complexity in Table 2 (i.e., the term $m \log r$). The query cost of PIR also significantly increases (especially the server cost) because it computes over all r blocks for answering each query. For SBR, although a larger r leads to a larger shuffling cost (e.g., increased communication in SBR-IBS and increased server computation in SBR-ROS) due to a larger row size n, the amortized shuffling cost remain reasonably small due to a large n_q shown in Table 5. This explains why both SBR still exhibit balanced communication and computation components in query cost and thus outperforms the baselines.

3) Effect of law parameter δ: A larger δ leads to a more skewed query distribution, therefore, triggers more frequent shuffles in SBR to enforce a uniform observed access distribution for long-term privacy. Figure 4 reports the query cost w.r.t. δ, while other parameters are fixed at default. In general, more skewed query distribution leads to increased query cost for SBR because of increased amortized shuffling cost. Typically, larger δ leads to the increase of communication cost in SBR-IBS and the increase of server computation cost in SBR-ROS, due to the adoption of communication-intensive (Interleave Buffer Shuffle) and server computation-insensitive (Repeatable Oblivious Shuffle) shuffling algorithms, respectively. However, skewed query distribution (e.g., $\delta = 1.2, 1.4$) does not destroy the balance between SBR's communication and computation costs. In contrast to Path ORAM and PIR (with static but expensive query cost), another advantage of SBR is that it can benefit from less biased query distribution ($\delta \leq 1.0$) for better performance.

4) Effect of block size m: A larger m indicates more data contained in each data block, thus, increases the query cost. Figure 5 reports the query cost w.r.t. m, while other parameters are fixed at default. As expected, all methods show an increase in query cost. In term of communication cost and client cost, SBR-IBS has better query performance than SBR-ROS with small m, whereas SBR-ROS outperforms SBR-IBS with large m. The reason is that Interleave Buffer Shuffle used in SBR-IBS suits for small-sized blocks but Repeatable Oblivious Shuffle used in SBR-ROS suits for large-sized blocks, as discussed in Section 2.2. The results verify the flexibility of SBR under different application scenarios.

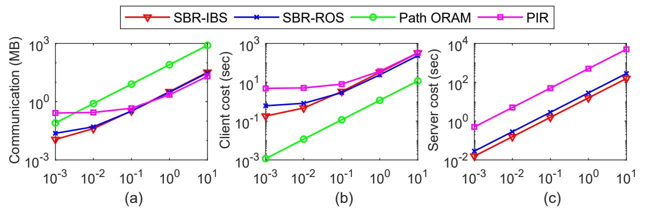


Figure 5: Query cost w.r.t. block size m (MB) (Path ORAM has no server computation and thus not reported in (c))

7 CONCLUSION

Efficient block retrieval from an outsourced database while protecting access patterns is a challenging problem. Existing PIR and ORAM solutions suffer from prohibitive computation or communication cost, and thus are impractical. To fix these performance bottlenecks, we present a new security definition, access pattern unlinkability, for measuring the protection of access patterns. This novel definition separates access pattern privacy into short-term privacy at individual query level and long-term privacy at query distribution level, which provides the flexibility to reconcile access pattern protection and query performance. Based on existing PIR and oblivious shuffling methods, we present the construction of the SBR protocol for secure block retrieval with access pattern unlinkability guarantees. Under different application scenarios, SBR can adopt different shuffling algorithms to have better query performance. Both analytical and empirical analysis demonstrate the flexibility and usability of SBR in practice.

REFERENCES

[1] Ben Adida and Douglas Wikström. 2007. How to shuffle in public. In *Proc. TCC'07.* 555.

[2] Ricardo Baeza-Yates. 2015. Incremental sampling of query logs. In *Proc. SIGIR'15.* ACM, 1093–1096.

[3] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proc. CCS'15.* 837–849.

[4] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2015. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)* 47, 2 (2015), 18.

[5] Yan Cheng Chang. 2004. Single database private information retrieval with logarithmic communication. In *Proc. ACISP'04.* 50.

[6] Zhao Chang, Dong Xie, and Li Feifei. [n.d.]. SEAL-ORAM. https://github.com/InitialDLab/SEAL-ORAM.

[7] Zhao Chang, Dong Xie, and Feifei Li. 2016. Oblivious ram: a dissection and experimental evaluation. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1113–1124.

[8] Guoxing Chen, Ten-Hwang Lai, Michael K Reiter, and Yinqian Zhang. 2018. Differentially private access patterns for searchable symmetric encryption. In *Proc. INFOCOM'18.* IEEE, 810–818.

[9] Ivan Damgård and Mads Jurik. 2001. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proc. PKC'01.*

[10] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion oram: A constant bandwidth blowup oblivious RAM. In *Proc. TCC'16.* 145.

[11] A Stewart Fotheringham and Daniel C Knudsen. 1987. *Goodness-of-fit statistics.* Number 46. Geo.

[12] Craig Gentry and Zulfikar Ramzan. 2005. Single-database private information retrieval with constant communication rate. In *Proc. ICALP'05.* 803.

[13] Michael T Goodrich. 2014. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(nlogn) time. In *Proc. STOC'14.* 684.

[14] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Practical oblivious storage. In *Proc. CODASPY'12.* 13–24.

[15] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-abuse attacks against order-revealing encryption. In *Proc. SP'17.* 655.

[16] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proc. SIGMOD'02.* 216.

[17] Perry R Hinton. 2014. *Statistics explained.* Routledge.

[18] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure multidimensional range queries over outsourced data. *VLDB Journal* 21, 3 (2012), 333–358.

[19] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proc. NDSS'12.* 12.

[20] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proc. CCS'16.* 1329.

[21] Weipeng Lin, Ke Wang, Zhilin Zhang, and Hong Chen. 2017. Revisiting security risks of asymmetric scalar product preserving encryption and its variants. In *ICDCS.* IEEE, 1116–1125.

[22] Helger Lipmaa. 2005. An oblivious transfer protocol with log-squared communication. In *Proc. ISC'05.* 314.

[23] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. 2006. l-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06).* IEEE, 24–24.

[24] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS'14.* 1–11.

[25] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proc. CCS'15.* 644.

[26] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *Proc. ICALP.* 556.

[27] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).* Springer, 223–238.

[28] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2017. CacheShuffle: An Oblivious Shuffle Algorithm Using Caches. *arXiv preprint arXiv:1705.07069* (2017).

[29] Casper Petersen, Jakob Grue Simonsen, and Christina Lioma. 2016. Power law distributions in information retrieval. *ACM Transactions on Information Systems (TOIS)* 34, 2 (2016), 8.

[30] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *Proc. CRYPTO'10.* 502.

[31] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Proc. SP'13.* 253–267.

[32] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[33] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path oram: An extremely simple oblivious ram protocol. In *Proc. CCS'13.* 299.

[34] Sen Su, Yiping Teng, Xiang Cheng, Ke Xiao, Guoliang Li, and JunLiang Chen. 2015. Privacy-preserving top-k spatial keyword queries in untrusted cloud environments. *IEEE Transactions on Services Computing* (2015).

[35] Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.

[36] Jun Tang, Yong Cui, Qi Li, Kui Ren, Jiangchuan Liu, and Rajkumar Buyya. 2016. Ensuring security and privacy preservation for cloud data services. *ACM Computing Surveys (CSUR)* 49, 1 (2016), 13.

[37] Yunchen Wu, Ke Wang, Zhilin Zhang, Weipeng Lin, Hong Chen, and Cuiping Li. 2018. Privacy preserving group nearest neighbor search. In *Proc. EDBT'18.* 277–288.

[38] Dong Xie, Guanru Li, Bin Yao, Xuan Wei, Xiaokui Xiao, Yunjun Gao, and Minyi Guo. 2016. Practical private shortest path computation based on oblivious storage. In *Proc. ICDE'16.* 361–372.

[39] Bin Yao, Feifei Li, and Xiaokui Xiao. 2013. Secure nearest neighbor revisited. In *Proc. ICDE'13.* 733.

[40] Xun Yi, Mohammed Golam Kaosar, Russell Paulet, and Elisa Bertino. 2013. Single-database private information retrieval from fully homomorphic encryption. *Transactions on Knowledge and Data Engineering* 25 (2013), 1125–1134.

[41] Zhilin Zhang, Ke Wang, Chen Lin, and Weipeng Lin. 2018. Secure top-k inner product retrieval. In *Proc. CIKM'18.*

[42] Zhilin Zhang, Ke Wang, Weipeng Lin, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2019. Repeatable Oblivious Shuffling of Large Outsourced Data Blocks. Cryptology ePrint Archive, Report 2019/071. https://eprint.iacr.org/2019/071.