An Efficient PIR Construction Using Trusted Hardware

Yanjiang Yang[1,2], Xuhua Ding[1], Robert H. Deng[1], and Feng Bao[2]

[1] School of Information Systems, Singapore Management University, Singapore 178902
[2] Institute for Infocomm Research, Singapore 119613

Abstract. For a private information retrieval (PIR) scheme to be deployed in practice, low communication complexity and low computation complexity are two fundamental requirements it must meet. Most existing PIR schemes only focus on the communication complexity. The reduction on the computational complexity did not receive the due treatment mainly because of its $\mathrm{O}(n)$ lower bound. By using the trusted hardware based model, we design a novel scheme which breaks this barrier. With constant storage, the computation complexity of our scheme, including offline computation, is linear to the number of queries and is bounded by $\mathrm{O}(\sqrt{n})$ after optimization.

1 Introduction

Private Information Retrieval (PIR) was first formulated by Chor et. al. in [5]. It offers a strong privacy assurance since it disallows any leakage of user query information. Although PIR should be the ideal privacy guardian of commercial database users, this did not happen. The reason is its prohibitively high cost, as pointed out by Sion and Carbunar [12]. Their analysis shows that a carefully designed PIR scheme with sophisticated cryptographic techniques costs even more time delay than the most trivial solution of transferring the entire database. The culprit for this unexpected effect is the expensive computation cost, which dominates the overall time delay. In the standard PIR model, the lower computation bound is obviously $\mathrm{O}(n)$ where n is the database size. A new model based on trusted hardware was introduced in [7, 8], which has a logarithm communication complexity and constant online computational complexity. Nonetheless, those schemes are not practical either, since they have to periodically shuffle the *entire* database. Considering the scale of modern databases, a full database shuffle disrupts the database service.

The objective of this paper is to narrow the gap between the ideality and the practicability of PIR. We construct a practical PIR scheme using the same trusted hardware model as in [7, 8, 14]. With a constant storage cost of the trusted hardware, our construction requires $\mathrm{O}(\log n)$ communication cost and $\mathrm{O}(\sqrt{n})$ computation cost per query including constant online computation and amortized offline computation.

Related Work Many PIR constructions were proposed to reduce the communication complexity, including [4, 11, 9, 10, 2]. As shown in [13, 7, 8, 14], the communication complexity can be reduced to $\mathrm{O}(\log n)$ by using a trusted hardware embedded in the database server. In this model, a trusted hardware is able to perform encryptions/decryptions and has a secret cache of reasonable size for storing retrieved data items. Further advantage of this type of schemes is the O(1) online computation cost for each query. However, all of them require a database re-encryption and re-shuffle whenever the cache is full. Since the available space in the cache decreases linearly with the number of queries, a full database shuffle is performed frequently which requires $O(n)$ operations.

The previous work focusing on computation cost reduction is [3], where Beimel et. al. proposed a new model called PIR with Preprocessing. This model uses k servers each storing a copy of the database. Before a PIR execution, each server computes and stores polynomially-many bits regarding the database. This approach reduces both the communication and computation cost to $O(n^{1/k+\epsilon})$ for any $\epsilon > 0$. However, it requires a storage of a polynomial of n bits, which is infeasible in practice. A recent scheme [15] improves the communication and computation complexity to $O(\log^2 n)$ with a cache storing $O(\sqrt{n})$ records.

We notice that the trusted hardware based PIR model is similar to the model in ORAM [6]. But we stress that the "square root" complexity in [6] and our result are in different context. The square root solution of ORAM requires a sheltered storage storing $\sqrt{n}$ words, which is equivalent to using a cache storing $\sqrt{n}$ items in the PIR model. Our scheme in this work, however, only uses a *constant* size cache.

Roadmap We define the system model and the security notion of our scheme in Section 2. A basic construction is presented in Section 3 as a steppingstone to the full-fledged scheme in Section 4. Performance of our scheme is discussed in Section 5, and Section 6 concludes the paper.

2 System Model and Definition

System Model Our scheme follows the trusted hardware model used in [8, 7, 14]. The architecture consists of a group of users, a database D modeled as an array of n data items of equal length denoted by $d_1, d_2, \cdots d_n$, respectively, and a database host H where D is stored. A trusted component denoted by T is embedded in H. To retrieve d_i, a user sends to T a query specifying the index i. T then interacts with H which operates over the encrypted database, and at the end of the execution, T returns d_i to the user. We assume that the communication channel between users and T is confidential.

T is a tamper-resistant hardware with a cache for storing up to k data items, $k \ll n$. No other entity (except T itself) is able to tamper T's protocol executions or access its private space including the cache. T is capable of performing certain cryptographic operations, such as symmetric key encryptions/decryptions and pseudo-random number generation. In practice T can be implemented by using a specialized trusted hardware such as IBM PCIXCC [1].

Security Definition The adversary in our model is the database host H, which has polynomial-bounded computational power, and attempts to derive information from the PIR protocol execution. The adversary is able to not only observe accesses to its space, including all read/write operations on the database, but also query the database like a legitimate user.

We use **access pattern** to describe the information observed by the adversary within a time period of protocol execution. When T accesses H's space including the memory and disks, H observes the data in use and the involved positions. The access pattern of length $m \geq 1$ is defined as a sequence of m elements $\langle \alpha_1, \cdots, \alpha_m \rangle$, where each α_i represents the information observed by H during an access to H's space. We use $\mathbb{A}_D$ to denote the set of all possible access patterns generated by querying the database D.

The security notion in ORAM [6] is used here to measure the information leakage from PIR query executions. A secure PIR scheme should ensure that the adversary does not gain additional information about the queries from the access pattern, except the a-priori information. This notion is similar to *perfect secrecy* defined for ciphers where an adversary obtains no additional information about the plaintext from the ciphertext. More formally, let Q be the random variable representing a user query, whose value is the index of the requested item, denoted by $q \in [1, n]$. $\Pr(Q = q)$, or simply $\Pr(q)$, denotes the probability that a query is on d_q. Then, the notion of privacy is defined as:

Definition 1. *A PIR scheme is secure, iff for every database of $n \in \mathbb{N}$ items, given a user query, for every valid access pattern, the conditional probability for the event that the query is on any index q is the same as its a-priori probability, i.e.* $\forall D = \{d_1, \cdots, d_n\}, \forall q \in [1, n], \forall \mathcal{A} \in \mathbb{A}_D, \Pr(Q = q|\mathcal{A}) = \Pr(Q = q)$. $\square$

3 Basic Construction

3.1 Overview

We briefly recall the idea of the schemes in [8, 7, 14] which are the predecessor of ours. During the system *initialization*, the database is encrypted and permutated by a trusted authority. All subsequent retrievals are operated upon the encrypted database. The database service is provided in *sessions*. A session starts when T's cache is empty and ends when it is full. During a session, T retrieves the requested item from the database if it is not in the cache; otherwise, a random item is read. At the end of the session, the entire database is shuffled, and then a new session commences. The objective of database shuffles is to re-mix the touched records within the database, so that the database host has no idea whether a record in the newly shuffled database has ever been read.

We observe that shuffling the entire database is not indispensable, as long as user queries generate access patterns of identical distribution. Based on this observation, we in in this work propose a new PIR scheme with *partial* shuffles, where only those records that have ever been accessed are shuffled. We also design a novel *twin retrieval* method, which forces user queries to generate access patterns of the same distribution. A conceptual view of the protocol execution is as follows. A record is labeled *black* if it has ever been accessed. Otherwise, it is *white*. During the system initialization, T generates a secret key sk for a semantically secure cipher, and a secret random permutation $\sigma : [1, n] \to [1, n]$. Every item d_i in D and its index i are encrypted under sk and written into the $\sigma(i)$-th position of D_0 as a record. In the rest of the paper, we refer to an entry in the original database D and its location as item and index, and refer to an entry in the encrypted database and its location as record and position. We use d_i to denote the i-th item in D, and a_i to denote the i-th record in the shuffled database. After D_0 is generated, all records in D_0 are initially white.

Our PIR service also proceeds in sessions, and the encrypted database in the s-th session is denoted by D_s. During a session, for each user query T executes a *twin retrieval*: if the requested item d_i is in the cache, T reads one random black record and one random white record from D_s; otherwise, T reads the corresponding record and reads one random record in a different color. After the cache is full, T then generates a new random secret permutation π_{s+1} for all black records and updates D_s into D_{s+1} by shuffling and re-encrypting all black records. Those white records remain intact. After the *partial shuffle*, H only knows that a black record has ever been read, but does not know in which session and how many times it has been accessed.

The key problem in implementing this approach is how T securely decides whether a record is black or white. While the label bits of the black records are set, T can not directly access H to check those bits since the access implicates that those records are sought by T. In the following, we assume that T's cache is big enough to accommodate the positions of all black records, so as to facilitate better understanding the idea of our new PIR approach. We remove this assumption in Section 4 by introducing an improved construction.

3.2 A Basic PIR Scheme

We use an array B to keep the black positions in an ascending order. If a_x is a black record and $B[i] = x$, we say that i is the B-Index of a_x. B is stored in H's space and maintained by H: whenever a record is read, it updates B. We use B_s to denote B's state in the beginning of the s-th session. T copies B into its cache before a session starts. During a session, B is updated, whereas T's copy is not changed. Note that for each record read into the cache, T needs to store the corresponding data item and its index in the cache. We denote the cache content by $\mathcal{C}$ and use $\mathcal{C}.Ind$ to denote the set of all stored indices.

A permutation π_s, $s \geq 1$, specifies the mapping between the sets of black positions in D_s and D_0. It is essentially a permutation of B-indexes of all black records. Let $\mathbb{Z}_{|B|} = \{1, 2, \cdots, |B|\}$. Formally, the permutation $\pi_s : \mathbb{Z}_{|B|} \to \mathbb{Z}_{|B|}$, is defined as: $\pi_s(i) = j$ *if and only if* $D_s[B_s[j]]$ *and* $D_0[B_s[i]]$ *contain the same item, which is* $D[\sigma^{-1}(B_s[i])]$. Note that σ is a mapping between all entries in D_0 and D. The relations among these notations are $D \overset{\sigma}{\Longrightarrow} D_0 \xrightarrow{\pi_s} D_s$. With B_s, π_s and σ, we are able to locate a record in D_s for a given item index. The PIR protocol proceeds in sessions shown below.

Session 0. T executes k queries using the retrieval algorithm in [14]. Specifically, for a query on the i-th item of D, $i \in [1, n]$, if the requested one is not in

T's cache $\mathcal{C}$, T reads the $\sigma(i)$-th record from D_0 into $\mathcal{C}$. Otherwise, T retrieves a random record. At the end of the session, T generates a new random secret permutation $\pi_1 : [1,k] \rightarrow [1,k]$. It shuffles the k black records according to π_1 while leaving the white records intact. Since all records to be shuffled are in $\mathcal{C}$, T simply re-encrypts and writes them out sequentially to generate D_1, and clears the cache.

Session $s \geq 1$. When session s starts, $\mathcal{C}$ is empty. T processes $k/2$ queries in the session. For a user query, T executes Algorithm 1 shown below. At the end of the session, T executes Algorithm 2 to shuffle all black records.

Algorithm 1 Basic Twin Retrieval Algorithm in Session $s \geq 1$. **Input:** a query on i, $i \in [1,n]$, $B_s[1,(s+1)k/2]$. **Output:** the item $d_i \in D$.

```
1: Through the secure channel, T accepts a query from the user requesting the $i$-th
   item in $D$.
2: if $i \notin \mathcal{C}.Ind$ then
3:   $j = \sigma(i)$.
4:   binary-search $j$ in $B_s$; /*we do not elaborate the binary-search algorithm since it
     is a standard one*/
5:   if exists $u$, s.t. $B_s[u] = j$ then
6:     $d_i$ is black; Read $D_s[B_s[\pi_s(u)]]$ as $d_i$ and read a random white record;
7:   else
8:     $d_i$ is white; read a random black record and read $D_s[j]$ as $d_i$;
9:   end if
10: else
11:   read a random black record and a white record from $D_s$ into $\mathcal{C}$.
12: end if
13: return $d_i$ to the user.
```

We now explain the retrieval algorithm (Algorithm 1) and the shuffle algorithm (Algorithm 2). In Algorithm 1, T searches B_s to determine the color of the requested record. For a white record, T directly uses its image under σ to read the data, since it has never been shuffled. For a black records, T computes its B-index under π_s and then looks up B_s to locate its position in D_s. Since B_s is inside T's cache, all accesses are not visible to the server. For a query execution, H only observes one read to a black record and one read to a white record. After $k/2$ queries, the cache is full, where half are black and half are white. B maintained by H now has $(2+s)k/2$ entries.

The partial shuffle is to mix the black records including those newly retrieved during the session, so that they are randomly relocated in D_{s+1}. The basic idea of the algorithm is the following: T updates the black positions in D_s sequentially. For each black position, T figures out the location of its preimage under π_{s+1}. If the preimage is in $\mathcal{C}$, T finds the next position whose preimage is not in $\mathcal{C}$ (as shown in Step 5, 6, 7). The computation of the preimage location involves the composition of π_{s+1}^{-1} and π_s. Since π_{s+1}^{-1}'s range is larger than π_s's domain, a translation from an index in B_{s+1} to B_s is needed (Step 8). As B_{s+1} is actually a combination of sorted B_s and the white positions (positions of newly retrieved white records) in sorted $\mathcal{C}$, we are ensured that $B_{s+1}[i] = B_s[i-\delta]$ (Step 8),

Algorithm 2 Basic Partial Shuffle Algorithm executed by T at the end of s-th session, $s \geq 1$. **Input:** B with $(2+s)k/2$ black records, cache $\mathcal{C}$ with $k/2$ black and $k/2$ white records; **Output:** D_{s+1}

```
1: scan $B$. For each item in the cache, calculate its index in $B$.
2: secretly generate a random permutation $\pi_{s+1} : \mathbb{Z}_{|B|} \rightarrow \mathbb{Z}_{|B|}$.
3: for $(i = i' = 1; i \leq sk/2; i++)$ do
4:     $j = \pi_{s+1}^{-1}(i')$;
5:     while $\sigma^{-1}(B_s[j]) \in \mathcal{C}.Ind$ and $i' \leq sk/2$ do
6:         $i' = i' + 1$; $j = \pi_{s+1}^{-1}(i')$; /*find one not from $\mathcal{C}$*/
7:     end while
8:     count $\delta$ as the number of white indexes in $\mathcal{C}$ which are smaller than $j$,
9:     compute $v = \pi_s(j - \delta)$; read $D_s[B_s[v]]$.
10:    if $i \neq i'$ then
11:        Re-encrypt $D_s[B_s[v]]$ into $D_{s+1}[B[i]]$;
12:    else
13:        Insert $D_s[B_s[v]]$ to cache. Retrieve the corresponding item from $\mathcal{C}$ and re-
           encrypt it to $D_{s+1}[B[i]]$.
14:    end if
15:    $i' = i' + 1$;
16: end for
17: write the remaining $k$ records in $\mathcal{C}$ to $D_{s+1}$ accordingly, securely eliminate $\pi_{s-1}$.
18: copy $B$ into the cache as $B_{s+1}$. End the session.
```

where δ is the number of white indices in $\mathcal{C}$ smaller than i. The average cost of finding δ is $\mathrm{O}(\log k)$ (The cost can be reduced to $\mathrm{O}(1)$ by keeping two copies of B in the cache and using pointers to link them.). Among the variables used in Algorithm 2, $B[i]$ points to the black position in D_{s+1} for writing whereas $B_s[i']$ points to the black position in D_s for reading. None of them decreases. Therefore, the overall complexity is $\mathrm{O}(sk \log k)$.

3.3 Security Analysis

Due to the length limit, we only formalize the security of our scheme by presenting the following lemmas, whose proofs are available in Appendix. Lemma 1 shows that the basic partial shuffle (Algorithm 2) is uniform in the sense that after the partial shuffle at the end of Session s, the previous black records in D_s and the white records retrieved during the session are randomly re-located to D_{s+1}. Thus, all black records appear indistinguishable to H. Then, Lemma 2 claims that at any time, the access patterns for any two queries of the basic twin retrieval algorithm (Algorithm 1) have the same distribution. Finally, by the results of Lemma 1 and Lemma 2, we prove in Theorem 1 that the basic PIR scheme is secure, satisfying Definition 1.

Lemma 1 (Uniform Shuffle). *The basic partial shuffle algorithm performs a uniform shuffle on all black records. Namely,* $\forall s > 0$, $\forall \jmath, i \in B_s$,

$$\Pr(D_s[\jmath] \simeq D_0[i] \,|\, \mathcal{A}_0, \mathcal{R}_0, \cdots, \mathcal{A}_{s-1}, \mathcal{R}_{s-1}) = 1/|B_s|,$$

where $\mathcal{A}_l$ and $\mathcal{R}_l$, $l \in [0, s-1]$ are the access pattern and the reshuffle pattern for the l-th session, respectively. $D_s[j] \simeq D_0[i]$ means $D_s[j]$ and $D_0[i]$ have the same plaintext.

Lemma 2 (Uniform Access). *Let Q be the random variable for the requested item's index in D. Let (X, Y) be the two-dimensional random variable for the positions of the black record and the white record accessed in the twin retrieval algorithm corresponding to Q. $\forall q_1, q_2 \in [1, n]$, suppose $\mathcal{A}$ is the access pattern when $Q = q_1$ or $Q = q_2$ is executed, then $\Pr((X = x, Y = y) \mid \mathcal{A}, Q = q_1) = \Pr((X = x, Y = y) \mid \mathcal{A}, Q = q_2)$.*

Theorem 1 (Security of PIR). *Let $\mathcal{A}_K$ be the access pattern of K database accesses. For query Q, $\forall q \in [1, n]$, $\forall K \in \mathbb{N}$, $\forall \mathcal{A}_K$, $\Pr(Q = q \mid \mathcal{A}_K) = \Pr(Q = q)$.*

4 A Construction Without Storage Assumption

In this section, we propose an improved scheme without assuming T's capability in storing B. As we mentioned earlier, the exposure of accesses to B leads to security breaches, since it indicates that the accessed ones are entries pertaining to the query in execution. Informally, the access to B requires a PIR-like solution. A trivial solution is to treat B as a database and to run a PIR query on it. Surely, the cost of this approach seriously counteracts our effort to improve the computational efficiency. We design a much more efficient solution due to the fact that T has the prior knowledge of those accesses.

4.1 Auxiliary Data Structures

Management of Black Positions. Recall that D_s is a result of a partial shuffle under the permutation $\pi_s : \mathbb{Z}_{|B|} \to \mathbb{Z}_{|B|}$. We use $|B|$ pairs of tuples $\langle x, y \rangle$ to represent this mapping, where $x \in \mathbb{Z}_n$ is a position in D_0 and $y \in \mathbb{Z}_n$ is the corresponding position under π_s in D_s. T selects a *deterministic* symmetric key encryption scheme $e(\cdot)$ and a secret key u. Let $f_u(x) = \mathcal{H}(e_u(x))$, where $\mathcal{H}$ is a hash function. These $|B|$ half-encrypted pairs are stored in an sorted array $L = [(f_u(x_1), y_1), (f_u(x_2), y_2) \cdots, (f_u(x_{|B|}), y_{|B|})]$, where $y_1 < \cdots < y_{|B|}$. Note that the sequence of y-values in L is exactly array B, which explains why we leave y-values in plaintext. However, B is updated by H due to query executions whereas L is not. We also build a *complete binary search tree Γ* where the tuples in L are the leaves in the following manner: from left to right, the leaves are sorted in an ascending order of $f_u(x)$ values. All the $|B| - 1$ inner nodes are integers randomly assigned by T according to their left and right children.

T makes use of L and Γ to decide whether an item is a white or black record, and to read a specific or random black record.

- To read an item with index x: If $f_u(\sigma(x))$ is smaller than the leftmost leaf or larger than the rightmost leaf of Γ, T immediately knows that $\sigma(x)$ is a white position. Otherwise, it runs a binary search for $f_u(\sigma(x))$ in Γ. Suppose

that the search ends at a leaf node $\langle f_u(x'), y\rangle$. If $f_u(x) = f_u(x')$, y is the position of the requested item; otherwise, y is taken as a randomly selected black position.
– Random search: Starting from the root, T tosses a coin at each level to select either the left child or the right child as the next node to read. In the end, it returns a leaf.

L and Γ are (re)constructed at the end of each session. L is initialized when T executes the partial shuffle under π_s whose algorithm is explained Section 4.2. During a shuffle, T sequentially writes to those positions stored in B, which is exactly y-values in L. Therefore, for each data item d_i relocated to the black position stored at $B[r]$, T sets $L[r] = \langle f_u(\sigma(i)), B[r]\rangle$, where u is a new encryption key. Once L is established, construction of Γ is straightforward.

Management of White Positions. We need to manage white records as well. The $|B|$ black records virtually divide the database into white segments, i.e. blocks of adjacent white records. We use an array $W[\,]$ in H's space to sequentially store these white segments, such that $W[i] = \langle l, m, M\rangle$ indicating that the i-th white segment in the database starts from the record a_l and contains m white records. We set $W[i].M = \sum_{j=1}^{i-1} W[j].m + 1$ such that a_l is the $W[i].M$-th white record in the database. Different from L and Γ, W is managed by H. T makes use of W to read white records in the following manner.

– To read the white record with index x: T runs a binary search on W for the address $\sigma(x)$, such that it stops at $W[i]$, such that $W[i].l \le \sigma(x) < W[i+1].l$. Then, it reads the $\sigma(x)$-th records from D_s.
– Random search: T generates $r \in_R [1, n - |B|]$. Then it runs a binary search on W for the r-th white record in D_s, such that it stops at $W[i]$, such that $W[i].M \le r < W[i+1].M$. Finally, it returns $y = W[i].l + r - W[i].M$.

For both cases, H only observes two search paths, which H cannot differentiate the two types of retrievals.

We need to store more information in $\mathcal{C}$ as well. Suppose that T retrieves a record a_j into $\mathcal{C}$. A new entry is created as a tuple (*BIndex, Color, Ind,Data*) where Ind and *Data* are the corresponding item's index and value, respectively; *Color* is set to 'B' if a_j was black before retrieval; otherwise Color is set to 'W'; *BIndex* is set to a_j's B-Index with respect to D_0. We use $\mathcal{C}[i]$ to denote the i-th entry of the cache, $\mathcal{C}.Ind$ to denote the set of all entries' *Ind* values, $\mathcal{C}.BIndex$ to denote the set of all entries' *BIndex* values. Note that B_s is no longer used, as B is not stored in $\mathcal{C}$.

4.2 The Improved Scheme

We are now ready to present the scheme. It proceeds in a similar way as the basic scheme in Section 3.2. The difference is that since Session 1, T executes Algorithm 3 for a query execution and Algorithm 4 for the partial shuffle. Algorithm 3 shows how to process a query during the s-th session, $s \ge 1$. Note that $|B| = (s+1)k/2$ when session s starts.

Algorithm 3 Improved Twin Retrieval Algorithm in Session $s \geq 1$, executed by T.
Input: a user query on $i \in [1, n]$. **Output:** the data item $d_i \in D$.

```
1: Through the secure channel, $\mathsf{T}$ accepts a query from the user requesting for the
   $i$-th item in $D$.
2: $min = B[1]$; $max = B[(s+1)k/2]$; /*$(min, max)$ is the range of black positions.*/
3: $i' = \sigma(i)$;
4: if $i \in \mathcal{C}.Ind$ then
5:    randomly search $\Gamma$ which returns $\langle f_u(x), y\rangle$. Then $j_b \leftarrow y$.
6:    randomly search $W$ which returns $j_w$.
7: else
8:    if $i' < min$ or $i' > max$ then
9:       randomly search $\Gamma$ which returns $\langle f_u(x), y\rangle$. Then $j_b \leftarrow y$.
10:   else
11:      search $\Gamma$ for $f_u(i')$ which returns $\langle f_u(x, y)\rangle$. Then $j_b \leftarrow y$.
12:   end if
13:   if $f_u(x) = f_u(i')$ then
14:      randomly search $W$ which returns $j_w$.
15:   else
16:      search $W$ for $i'$, which returns $\langle l, m, M\rangle$. Then $j_w \leftarrow i'$.
17:   end if   /*Note that $f_u()$ is deterministic.*/
18: end if
19: read the $j_b$-th and the $j_w$ records from $D_s$, and creates two new entries for them
    $\mathcal{C}$ accordingly; return $d_i$ to the user.
```

Access Pattern of Retrievals We use $\mathcal{A}_s$ to denote the access pattern produced by Algorithm 3. There are three types of accesses to H's space. The first type is the database accesses. For simplicity, we use the accessed black and white positions, denoted by (α_i, α'_i), as the access pattern in the i-th query execution. The second type is the accesses to W during the searches. The output of a binary search on W determines the involved search path. Furthermore, the output of a search can be derived by observing the subsequent database access. Therefore, the second type of accesses does not reveal extra information. We do not include it in $\mathcal{A}_s$. The third type is the retrieval of elements in Γ. Similar to the previous reasoning, the access to Γ does not divulge extra information and is not included either. According to Algorithm 3, totally $k/2$ queries are executed in a session. Thus, the access pattern produced during the s-th session is $\mathcal{A}_s = \langle \alpha_1, \alpha'_1, \cdots, \alpha_{k/2}, \alpha'_{k/2}\rangle$. The access pattern in Session 0 is an exception, since one record is retrieved per query: $\mathcal{A}_0 = \langle \alpha_1, \cdots, \alpha_k\rangle$.

Algorithm 4 shows how to perform a partial shuffle at the end of the s-th session. Note B has expanded from $(s+1)k/2$ elements to $(s+2)k/2$ elements due to the $k/2$ retrievals of white records in this session. The partial shuffle process requires $(s+2)k/2$ database writes and $sk/2$ database reads. We remark that the computation cost for constructing Γ' is not expensive for the following two reasons. First, those operations are memory based integer comparisons, which are much cheaper than database accesses. Second, the sorting process can be done by H.

Algorithm 4 Improved Partial Shuffle Algorithm executed by T at the end of s-th session, $s \geq 1$. **Input:** cache $\mathcal{C}$ with $k/2$ black and $k/2$ newly retrieved white records; **Output:** D_{s+1}, Γ' and L'

```
1: scan $B$ and assign the *BIndex* field for each entry in $\mathcal{C}$. Specifically, for every $1 \leq b \leq |B|$, if $\exists a \in [1,k]$, s.t. $\sigma(\mathcal{C}[a].Ind) = B[b]$, then set $\mathcal{C}[a].BIndex = b$.
2: generate a secret random permutation $\pi_{s+1} : \mathbb{Z}_{|B|} \to \mathbb{Z}_{|B|}$, and a new encryption key $u'$.
3: for $(i = i' = 1; i \leq sk/2; i++)$ do
4:    $j = \pi_{s+1}^{-1}(i')$;
5:    while $\sigma^{-1}(B[j]) \in \mathcal{C}.Ind$ and $i' \leq sk/2$ do
6:       $i' = i' + 1$; $j = \pi_{s+1}^{-1}(i')$; /*find one not from $\mathcal{C}$*/
7:    end while
8:    count $\delta$ as the number of white indexes in $\mathcal{C}$ which are smaller than $j$,
9:    compute $v = L[\pi_s(j - \delta)].y$; Read $D_s[v]$. Suppose that $D_s[v] = E_{sk}(x, d_x)$.
10:   if $i' = i$ then
11:      Re-encrypt $D_s[v]$ into the $D_{s+1}[B[i]]$;
12:      $L'[i] \leftarrow \langle f_{u'}(\sigma(x)), B[i] \rangle$;
13:   else
14:      insert a 4-tuple $\langle 0, \text{'B'}, x, d_x \rangle$ into $\mathcal{C}$.
15:      find $l \in [1,k]$ satisfying $\mathcal{C}[l].BIndex = \pi_{s+1}^{-1}(i)$. Insert $E_{sk}(\mathcal{C}[l].Ind, \mathcal{C}[l].Data)$ to $D_{s+1}[B[i]]$.
16:      $L'[i] \leftarrow \langle f_{u'}(\sigma(\mathcal{C}[l].Ind)), B[i] \rangle$.
17:   end if
18:   i'=i'+1
19: end for
20: write the remaining $k$ records in the cache to $D_{s+1}$ and assign $L'$ accordingly.
21: construct $\Gamma'$ based on $L'$ and discard $\pi_s, L, \Gamma$.
```

Access Pattern of Shuffles We use $\mathcal{R}_s$ to denote the access pattern produced by Algorithm 4 at the end of the s-th session. There are three types of accesses. The first type is the accesses to B. However, since all accesses to B are in a predetermined order, they do not leak any information (they can be generated correctly by H without observing the execution). We exclude them from $\mathcal{R}_s$.

The second type is the read and write accesses to the database. According to our algorithm, a read access is always followed by a write access. The sequence of the writes are known to H before the shuffle, since it follows the sequence of positions in B. Furthermore, the contents of the writes are new ciphertext under a semantic secure encryption. Therefore, the access pattern of writes does not expose information to H. Considering the read pattern only, we use α_i, the position of the i-th read access, to represent the access pattern.

The third type of accesses is the read operations on L and the write operations on L'. Every write to L' is always preceded by a read access to the database. Moreover, the sequence of writings to L' and the contents of L', except those encryptions, can be determined by H without observing the execution. Therefore, the write accesses on L' do not leak information. Every read operation on L exposes the touched index of L and a black position y. However, the exposed black position y can also be determined by observing the subsequent database read. Since L is known to H, knowing the black position y naturally implies the knowledge of its position in L. Thus, it suffices to represent the access pattern only using the database accesses, i.e. $\mathcal{R}_s = \langle \alpha_1, \cdots, \alpha_{sk/2} \rangle$, where $\alpha_i \in [1, n], 1 \leq i \leq sk/2$.

The shuffle at the end of Session 0 is a special case, where all records to be shuffled are in T's cache. T simply writes them out to the corresponding positions following the permutation π_1, in which case, $\mathcal{R}_0 = \emptyset$.

4.3 Security Analysis

Security analysis of the improved scheme is based on that of the basic scheme. By Lemma 3 and Lemma 4, we show that the improved partial shuffle algorithm (Algorithm 4) and the improved twin retrieval algorithm (Algorithm 3) also perform a uniform shuffle and a uniform access, respectively. This in turn implies that Theorem 1 also holds for the improved scheme.

Lemma 3. *Lemma 1 also holds for the improved partial shuffle algorithm (Algorithm 4).*

Proof (sketch) We compare the access patterns of the improved scheme with those of the basic scheme. The analysis in Section 4.1 has shown that the accesses to Γ and L do not leak extra information. Both shuffle patterns have the same distribution, since they are only determined by the permutations in use. Thus the proof for Lemma 1 is also valid for Algorithm 4. □

Lemma 4. *Lemma 2 also holds for the improved twin retrieval algorithm (Algorithm 3).*

Proof (sketch) The only difference between patterns generated by Algorithm 1 and Algorithm 3 is that the latter uses the search of Γ to generate a random black record. Nonetheless, under the random oracle mode, the function $f_u(\cdot)$ outputs a random number. □

5 Scheme Complexity

The communication cost of our scheme remains the same as other hardware-based PIR schemes [8, 7, 14]. Namely, it requires $\mathrm{O}(\log n)$ communication complexity, as the user inputs a $\log n$-bit long index of the desired data item, and T returns exactly one item of constant length. The database read/write are counted as a part of the computation cost. Note that $\mathrm{O}(\log n)$ is the lower bound of communication complexity for any PIR construction.

When considering the computational complexity, we regard an access to the host H's space and a decryption/encryption as one unit of operation. In the s-th session, a query retrieval using Algorithm 3 requires $\mathrm{O}(\log(sk))$ operations due to the task of a binary search in Γ. A partial shuffle at the end of the s-th session requires $\mathrm{O}(sk \log k)$ operations. Thus, the overall computation cost in all s-th sessions is $\mathrm{O}(s^2 k \log k)$ for totally $(2+s)k/2$ query executions. Consequently, the average cost per query for s sessions is $\mathrm{O}(s \log k)$, which is independent of the database size.

When the session number s reaches the order of n, the advantage of our scheme diminishes. A remedy to this problem is to shuffle the entire database at the end of the t-th session. The full shuffle resets the system to its initial state (Session 0). All records are colored white again, as the traces of all previous accesses are removed. Since an early full shuffle might not be able to fully exploit the benefits of partial shuffles, it is necessary to determine an optimal t. Recall that a full shuffle costs $\mathrm{O}(n)$ operations. With a full shuffle for every t sessions, the total cost for s sessions becomes $\mathrm{O}((t^2 k \log k + n)s/t)$ and the average cost per query is $\mathrm{O}(t \log k + n/tk)$ which is minimal when $t \log k = n/tk$. Therefore, the optimal value for t is $\sqrt{\frac{n}{k \log k}}$. The cost per query becomes $\mathrm{O}(\sqrt{\frac{n \log k}{k}})$.

A comparison of our scheme against other PIR schemes is given in Table 1. Note that all previous hardware-based schemes [7, 8, 13, 14] claim $\mathrm{O}(1)$ computation complexity since they only count the cost of database accesses. In fact, all of them requires $\mathrm{O}(\log k)$ operations to determine if an item is in cache. Our scheme also has $\mathrm{O}(1)$ database read/write, though we need an additional cost for a binary search in Γ. For those PIR schemes without using caches, the computation cost per query is at least $\mathrm{O}(n)$. From the table, it is evident that our scheme substantially outperforms all other PIR schemes in terms of average query cost by paying a slightly higher price of online query processes.

Table 1. Comparison of Computation Performance.

Schemes	**Cost of online query process**	**Average overall cost per query**
Our scheme	$\mathrm{O}(\log s + \log k)$	$\mathrm{O}(s \log k)$
Our scheme with full-shuffles	$\mathrm{O}(\log s + \log k)$	$\mathrm{O}(\sqrt{\frac{n \log k}{k}})$
Scheme in [14]	$\mathrm{O}(1)$	$\mathrm{O}(n/k)$
Scheme in [7, 8]	$\mathrm{O}(1)$	$\mathrm{O}(\frac{n \log n}{k})$
Scheme in [13]	$\mathrm{O}(1)$	$\mathrm{O}(n)$
Other PIR schemes without using caches	$\mathrm{O}(n)$	$\mathrm{O}(n)$

Notations: n: the database size; s: the number of sessions; k: the size of the cache, $k << n$.

6 Conclusion

All existing PIR schemes have $\mathrm{O}(n)$ computational cost for each query. In this paper, we broke this barrier using a novel approach for database retrieval and shuffle. The average cost per query is reduced to $\mathrm{O}(s)$ where s is the number of queries, or $\mathrm{O}(\sqrt{n})$ in maximum if an optimization is used. We proved the security of our scheme.

Acknowledgement

This research is partly supported by the Office of Research, Singapore Management University.

References

1. Arnold, T., and Doorn, L. V. The ibm pcixcc: A new cryptographic coprocessor for the ibm eserver. *IBM Journal of Research and Development 48* (May 2004).
2. Beimel, A., Ishai, Y., Kushilevitz, E., and Raymond, J.-F. Breaking the $o(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of IEEE FOCS'02* (2002), pp. 261–270.
3. Beimel, A., Ishai, Y., and Malkin, T. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of CRYPTO'00* (2000), pp. 55–73.
4. Chor, B., and Gilboa, N. Computationally private information retrieval. In *Proceedings of the 29th STOC'97* (1997), pp. 304–313.
5. Chor, B., Kushilevitz, E., Goldreich, O., and Sudan, M. Private information retrieval. In *Proceedings of IEEE FOCS'95* (1995), pp. 41–51.
6. Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *Journal of the ACM 43*, 3 (1996), 431–473.
7. Iliev, A., and Smith, S. Private information storage with logarithm-space secure hardware. In *Proceedings of International Information Security Workshops* (2004), pp. 199–214.
8. Iliev, A., and Smith, S. Protecting client privacy with trusted computing at the server. *IEEE Security & Privacy 3*, 2 (2005), 20–28.
9. Kushilevitz, E., and Ostrovsky, R. Replication is not needed: single database, computationally private information retrieval. In *Proceeding of the 38th IEEE FOCS'97* (1997), pp. 364–373.
10. Kushilevitz, E., and Ostrovsky, R. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In *Proceedings of Eurocrypt'00* (2000), pp. 104–121.
11. Ostrovsky, R., and Shoup, V. Private information storage. In *Proceedings of the 29th STOC'97* (1997), pp. 294 – 303.
12. Sion, R., and Carbunar, B. On the computational practicality of private information retrieval. In *Proceedings of NDSS'07* (2007).
13. Smith, S., and Safford, D. Practical server privacy with secure coprocessors. *IBM Systems Journal 40*, 3 (2001), 683–695.
14. Wang, S., Ding, X., Deng, R., and Bao, F. Private information retrieval using trusted hardware. In *Proceedings of the 11th ESORICS'06* (2006), pp. 49–64.
15. Williams, P., and Sion, R. Usable PIR. In *Proceedings of NDSS 2008.*

Appendix

Proof of Lemma 1: (Sketch) This proof is nearly the same as Lemma 1 in [14] with the only difference being what records to be shuffled. An intuitive explanation is that we can treat the black record set pointed by B_s as a database in

[14]. There exist two critical points of the proof. 1) For any record in the cache, the probability of a black position in D_s being chosen as the shuffle destination is exactly $\frac{1}{B_s}$. In other words, its image position in B_s is uniformly selected. This is obvious since its is determined by a random π_s. As it is written from the cache, the selection of the position is independent of those access patterns. 2) For any record not in the cache, its preimage position in the previous shuffle was uniformly chosen from black positions in D_{s-1} pointed by B_{s-1}. This is addressed by using induction on s. $\square$

Proof of Lemma 2: (Sketch) Assume that Q is executed at session s. We prove the theorem by examining the cases when $s = 0$ and $s \geq 1$.
I: $s = 0$. The theorem clearly holds as D_0 is a random permutation of D. Therefore, for each instance of Q on D, its image Y on D_0 is uniformly distributed. X is always 0.
II: $s \geq 1$. According to the algorithm, for a query Q, a black record and a white record are read. Define $I = \{i | i \in [1, n], i \in C\}$ containing the indices whose corresponding items are in the cache, and $J_1 = \sigma^{-1}(B_s) \setminus I$ containing the indices of black records, but presently not in the cache, and $J_2 = [1, n] \setminus (I \cup J_1)$, containing the indices whose corresponding items have never been accessed so far. To prove the theorem, it is sufficient to demonstrate that for any $q \in [1, n]$, $\Pr((X = x, Y = y) \mid \mathcal{A}, Q = q)$ remains the same in the following cases covering all possibilities of q.

- Case (1) $q \in J_1$. T reads the corresponding black record and a random white record from D_s. Due to Lemma 1, the corresponding record could be in any position in B_s with the same probability. Therefore $\Pr(X = x \mid \mathcal{A}, q) = \frac{1}{|B_s|}$. Y is a random retrieval, which is independent of $\mathcal{A}$. Therefore, $\Pr((X = x, Y = y) \mid \mathcal{A}, Q = q) = (\frac{1}{|B_s|}, \frac{1}{n-|B_s|})$.
- Case (2) $q \in J_2$. T reads a random black record and the corresponding white record from D_s. The position of the white records is determined by σ. Therefore, $\Pr(Y = y \mid \mathcal{A}, q) = \frac{1}{n-|B_s|}$. X is a random retrieval independent from $\mathcal{A}$. Therefore $\Pr((X = x, Y = y) \mid \mathcal{A}, Q = q) = (\frac{1}{|B_s|}, \frac{1}{n-|B_s|})$.
- Case (3) $q \in I$. Both X and Y are randomly retrieved. So $\Pr((X = x, Y = y) \mid \mathcal{A}, Q = q) = (\frac{1}{|B_s|}, \frac{1}{n-|B_s|})$

This completes the proof. $\square$

Proof of Theorem 1: (Sketch) It is equivalent to prove that $\forall K \in \mathbb{N}$, $\Pr(\mathcal{A}_K \mid Q = q) = \Pr(\mathcal{A}_K)$. Fix a session s, we prove the theorem by using induction on the size of $\mathcal{A}_K$.
I: When $K = 1$, our target is to prove that $\Pr(X = x, Y = y \mid Q = q) = \Pr(X = x, Y = y)$, $\forall x, y \in [n]$.

$\Pr(x, y) = \sum_{i=1}^{n} \Pr(x, y | i) \Pr(i)$. Consider $\Pr(x, y \mid i)$. There are two cases:

- The record corresponding to i is in B_s. Therefore, $\Pr(x) = \frac{1}{|B_s|}$, due to the initial permutation; $\Pr(y) = \frac{1}{n-|B_s|}$ due to random access.

– The record corresponding to i is in $[n]\backslash B_s$. Therefore, $\Pr(x) = \frac{1}{|B_s|}$, due to random access; $\Pr(y) = \frac{1}{n-|B_s|}$ due to the initial permutation.

Thus, in both case $\Pr(x, y \mid i) = (\frac{1}{|\Delta|}, \frac{1}{n-|\Delta|})$ for both cases. Obviously, $\Pr(x, y \mid i) = \Pr(x, y \mid j)$ for all $i, j \in [1, n]$. Consequently, $\Pr(x, y) = \Pr(x, y \mid q) \sum_{i=1}^{n} \Pr(i) = \Pr(x, y \mid q), \forall q \in [1, n]$.

II: Suppose that when $K = k-1$, the equation holds. We then prove that it still holds when $K = k$, i.e. $\Pr(\mathcal{A}_k \mid Q = q) = \Pr(\mathcal{A}_k)$. Without loss of generality, let $\mathcal{A}_k = \mathcal{A}_{k-1} \cup (x, y)$, where $(X = x, Y = y)$ is the k-th database read.

$$\begin{aligned}
&\Pr(\mathcal{A}_{k-1}, (x, y) \mid q) = \Pr(\mathcal{A}_k) \\
&\frac{\Pr(\mathcal{A}_{k-1}, x, y, q)}{\Pr(q)} = \Pr(\mathcal{A}_{k-1}, (x, y)) \\
&\frac{\Pr(x, y \mid \mathcal{A}_{k-1}, q)\Pr(\mathcal{A}_{k-1}, q)}{\Pr(q)} = \Pr(\mathcal{A}_{k-1}, (x, y)) \\
&\Pr(x, y \mid \mathcal{A}_{k-1}, q) = \frac{\Pr(\mathcal{A}_{k-1}, x, y)}{\Pr(\mathcal{A}_{k-1} \mid q)} \\
&\Pr(x, y \mid \mathcal{A}_{k-1}, q) = \Pr(x, y \mid \mathcal{A}_{k-1}) \\
&(\because \text{ induction assumption})
\end{aligned}$$

Note that there are three exclusive cases for $Q = q$.

1. $Q = q$ occurs after the k-th database access;
2. $Q = q$ is the query for the k-th database access;
3. $Q = q$ occurs prior to the k-th database access.

We proceed to prove that the above equation holds for all three different cases.

Case 1: Obviously, in this scenario, $\mathcal{A}_{k-1}$ and (x, y) are independent of $Q = q$. Therefore, $\Pr(x, y \mid \mathcal{A}_{k-1}, q) = \Pr(x, y \mid \mathcal{A}_{k-1})$.

Case 2: Note that

$$\Pr(x, y \mid \mathcal{A}_{k-1}) = \sum_{q=1}^{n} \Pr(x, y \mid \mathcal{A}_{k-1}, q)\Pr(q \mid \mathcal{A}_{k-1})$$

, where $Q = q$ is the query corresponding (x, y). Due to Lemma 2, $\Pr(x, y \mid \mathcal{A}_{k-1}, q) = \Pr(x, y \mid \mathcal{A}_{k-1}, q')$, $\forall q, q' \in [1, n]$. Therefore,

$$\Pr(x, y \mid \mathcal{A}_{k-1}) = \Pr(x, y \mid \mathcal{A}_{k-1}, q) \sum_{i=1}^{n} \Pr(i \mid \mathcal{A}_{k-1})$$

. According to the induction, $\Pr(i \mid \mathcal{A}_{k-1}) = \Pr(i)$, we have $\Pr(x, y \mid \mathcal{A}_{k-1}) = (x, y \mid \mathcal{A}_{k-1}, q)$

CASE 3: Let Q' be the random variable for the k-th query which generates (x, y). Considering all possible values of Q', denoted by q', we have

$$\Pr(x, y|\mathcal{A}_{k-1}, q) = \sum_{q'=1}^{n} \Pr(x, y \mid \mathcal{A}_{k-1}, q, q')\Pr(q' \mid \mathcal{A}_{k-1}, q)$$

Note that $\Pr(x, y \mid \mathcal{A}_{k-1}, q, q') = \Pr(x, y \mid \mathcal{A}_{k-1}, q')$ since (x, y) is determined by Q' and $\mathcal{A}_{k-1}$ according to our PIR algorithm. Therefore,

$$\Pr(x, y|\mathcal{A}_{k-1}, q) = \sum_{q'=1}^{n} \Pr(x, y \mid \mathcal{A}_{k-1}, q')\Pr(q' \mid \mathcal{A}_{k-1}, q)$$

Since $Q' = q'$ is independent of $\mathcal{A}_{k-1}$ and $Q = q$, thus

$$\begin{aligned}\Pr(x, y|\mathcal{A}_{k-1}, q) &= \sum_{q'=1}^{n} \Pr(x, y \mid \mathcal{A}_{k-1}, q')\Pr(q' \mid \mathcal{A}_{k-1}) \\ &= \Pr(x, y \mid \mathcal{A}_{k-1})\end{aligned}$$

$\square$