Database Access Pattern Protection Without Full-shuffles

Xuhua Ding[1] Yanjiang Yang[2] Robert Deng[1]
[1]School of Information Systems, Singapore Management University
[2]Institute of Infocomm Research, Singapore

***Abstract*—Privacy protection is one of the fundamental security requirements for database outsourcing. A major threat is information leakage from database access patterns generated by query executions. The standard Private Information Retrieval (PIR) schemes, which are widely regarded as theoretical solutions, entail $\mathrm{O}(n)$ computational overhead per query for a database with n items. Recent works in [9], [10], [19] propose to protect access patterns by introducing a trusted component with constant storage size. The resulting privacy assurance is as strong as PIR. Though with $\mathrm{O}(1)$ online computation cost, they still have $\mathrm{O}(n)$ amortized cost per query due to periodically full database shuffles. In this paper, we design a novel scheme in the same model with provable security, which only shuffles a portion of the database. The amortized server computational complexity is reduced to $\mathrm{O}(\sqrt{n \log n/k})$. With a secure storage storing thousands of items, our scheme can protect the access pattern privacy of databases of billions of entries, at a lower cost than those using ORAM-based poly-logarithm algorithms.**

I. Introduction

In database applications, a malicious database server can derive sensitive information about user queries, simply by observing the database access patterns, e.g. the records being retrieved or frequent accesses to "hot" records. Such a threat is aggravated in the Database-as-a-Service (DaaS) model whereby a data owner outsources her database to an untrusted service provider. The concern on potential privacy exposure becomes a hurdle to the success of DaaS and other data oriented applications in cloud-like settings. Note that database encryption does not entirely solve the problem, because access patterns also include the visited addresses and the frequency of accesses.

Private Information Retrieval (PIR) formulated in [6] is the well-known cryptographic mechanism inhibiting information leakage from access patterns. Modeling the database service as a bit retrieval from a bit array in plaintext, PIR disallows a server to infer any additional information about queries. Many PIR schemes [5], [14], [11], [12], [2] have been proposed with the emphasis on lowering the communication complexity between the server and the user. Nonetheless, as pointed out by Sion and Carbunar [17], those PIR schemes incur even more turnaround time than transferring the entire database as a reply to the user, because the heavy computation incurred at the server outweighs the saved communication expense.

The computation cost can be greatly reduced by embedding a trusted component (e.g. a tamper-resistant device) at the server's end. Such PIR schemes[1] were initially introduced in [9], [10] based on the square-root algorithm proposed in the seminal work on Oblivious RAM [8]. Compared with the standard PIR schemes, these PIR schemes [9], [10] deal with *encrypted* data records rather than bits in plaintext. The assistance of a trusted component cuts off the turnaround time, though the asymptotic computation complexity remains at $\mathrm{O}(n)$. In this paper[2],we follow this line of research and design a novel PIR scheme, which requires $\mathrm{O}(\log n)$ communication cost, $\mathrm{O}(1)$ runtime computation cost and $\mathrm{O}(\sqrt{n \log n/k})$ overall amortized computation cost per query, where k is the trusted cache size.

A. Related Work

Many PIR constructions [5], [14], [11], [12], [2] considers unencrypted database with the main objective being improving the server-user *communication* complexity, rather than *server computation complexity*. The best known results are due to [13] with $\mathrm{O}(\log^2 n)$ *communication* cost. The construction is built on the length-flexible additively homomorphic public key encryption (LFAH) [7], without the support of trusted hardware. Note that its computation cost remains as $\mathrm{O}(n)$.

A notable effort focusing on computation cost reduction without a trusted hardware 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 $\mathrm{O}(n^{1/k+\epsilon})$ for any $\epsilon > 0$. However, it requires a storage of a polynomial of n bits, which is infeasible in practice.

Oblivious RAM [8] was initially proposed to protect a software's memory access pattern. It proposed two algorithms: a shuffle-based algorithm (a.k.a square-root algorithm) and a hierarchy-based algorithm. The former costs $\mathrm{O}(\sqrt{n} \log n)$ memory access for one original data access and requires $\mathrm{O}(n+\sqrt{n})$ of storage, whereas the latter has $\mathrm{O}(\log^3 n)$ access cost and requires $\mathrm{O}(n \log n)$ storage.

The shuffle-based algorithm inspires Smith et. al. to design a PIR scheme [18] with $\mathrm{O}(\log n)$ communication cost and

[1]Strictly speaking, they are not PIR as defined in [6] because they handle encrypted data rather than the plaintext. Nonetheless, they offer the same strength of privacy assurance as the standard PIR.

[2]The paper is a full version of [22].

$O(n \log n)$ computation cost (i.e. server accesses) for periodical shuffles, where a trusted hardware plays the role of CPU in ORAM and caches a constant number of data. This hardware-based PIR scheme was further investigated in [9], [10], [19]. The main algorithmic improvement was due to [19], which proposed an $O(n)$ shuffle algorithm. Therefore the amortized computation complexity is $O(n/k)$ where the hardware store k records.

The hierarchical algorithm also has several derivatives. Williams and Sion [20] reduced the computation complexity to $O(\log^2 n)$ by introducing $O(\sqrt{n})$ storage at the client side. The complexity was further improved to $O(\log n \log\log n)$ in [21] by using an improved sort algorithm with the same amount of client side storage. Recently, Pinkas and Reinman propose a more efficient ORAM in [15]. It achieves $O(\log^2 n)$ complexity with $O(1)$ client end storage. Though asymptotically superior, all these big-O complexity notations carry large constant factors. The complexity of the original ORAM has a coefficient larger than 6000 and the complexity of Pinkas and Reinman's scheme has a constant factor falling between 72 and 160. Therefore, if the database is not large (e.g. $n = 2^{20}$), these hierarchy based algorithms are not necessarily more efficient than the shuffle-based algorithms.

CAVEAT. The algorithms proposed in this paper belongs to the square-root algorithm [8] family, i.e. based on shfufles. A detailed comparison between our scheme and the state-of-the-art hierarchy-based ORAM [15] is presented in Section V. In addition, we stress that the"square root" complexity of the shuffle-based ORAM and our results are completely in different context. The square root solution of ORAM requires a sheltered storage storing $\sqrt{n}$ items, which is equivalent to using a cache storing $\sqrt{n}$ items at the client end in our setting. In fact, our scheme only uses a *constant* size cache and when $k = \sqrt{n}$ our scheme has poly-logarithm complexity.

Roadmap: We define the system model and the security notion of our scheme in Section II. A basic construction is presented in Section III as a steppingstone to the full-fledged scheme in Section IV. Performance of our scheme is discussed in Section V, and Section VI concludes the paper.

II. SYNOPSIS

A. System Model

The system consists of a group of users, a database D modeled as an array of n data items of equal length denoted by $\{d_1, d_2, \cdots d_n\}$, and a database host denoted by H. A trusted component[3] denoted by T is embedded in H. T has an internal cache which stores up to k data items, $k \ll n$. No adversary can tamper T's executions or access its private space including the cache. T is capable of performing symmetric key encryption/decryption and pseudo-random number generation. All messages exchanged between users and T are through a confidential and authentic channel.

A PIR scheme in this model is composed of two algorithms: a shuffle algorithm and a retrieval algorithm. The former permutes and encrypts D while the latter executes PIR queries. The scheme runs in *sessions*. The database used in the s-th session is denoted by D_s, which is a permuted and encrypted version of D and is also stored in H's space. Within the session, T runs the retrieval algorithm to execute a PIR query, which involves fetching D_s records to its cache. The session ends when the cache is full. Then, T runs the shuffle algorithm which empties the cache and produces D_{s+1}. Note that D is never accessed by T.

a) Notations and Terminology: To highlight the difference between D and D_s, we use *item* to refer to any entry in D and use *record* to refer to any entry in D_s. We say that i is the *index* of d_i in D, and use *address* to refer to a record's position in D_s. A PIR query Q on item d_i is denoted $Q = i$, and we say that i is the *value* of Q. A summary of all notations and terms used in the paper is presented in Table I below.

Notations & Terms	Description
$D_s[i] \simeq D_{s'}[j]$	the decryptions of $D_s[i]$ and $D_{s'}[j]$ are the same data item.
item d_i, index	the i-th entry in the original database D. i is the index of d_i.
record $D_s[x]$, address	the x-th entry in D_s. A record is the ciphertext of an item. x is the address of $D_s[x]$.
$B[\,]$	the array of addresses of all black records, sorted in ascending order.
$\sigma : [1,n] \to [1,n]$	the initial permutation used for shuffling D into D_0. Item d_i is encrypted and shuffled to the $\sigma(i)$-th record in D_0.
$\pi_s : [1,\lvert B\rvert] \to [1,\lvert B\rvert]$	the permutation used in the s-th session. It defines the mapping between D_s and D_0. Its domain is decided by the size of B in the s-th session.
k	the maximum amount of items stored in T's cache.

TABLE I
TABLE OF NOTATIONS AND TERMS

B. Security Model

In a nutshell, a PIR scheme prevents an adversary from inferring information about queries from observation of query executions. The transcript of protocol execution within a period is referred to as *access pattern*. We use λ_K to denote an access pattern of length K. More formally, $\lambda_K = \{(a_j, D_{i_j}[a_j])\}_{j=1}^{K}$, where a_j is an address of database D_{i_j} and $D_{i_j}[a_j]$ is the a_j-th record in D_{i_j}. When D_{i_j} can be inferred from the context, we only use a_j to represent an access just for the sake of simplicity.

The adversary in our model is the database host H which attempts to derive information about user queries from access patterns. Besides observing all accesses to its memory or hard disk, H can also adaptively initiates PIR queries of its choices. Formally, we model the adversary as a probabilistic polynomial time algorithm $\mathcal{A}$, which takes any access pattern as the input and outputs the value of a target query. We allow $\mathcal{A}$ to access a query oracle $\mathcal{O}$, through which $\mathcal{A}$ issues PIR queries arbitrarily as a regular user and observes their executions.

Since the adversary can issue queries, we differentiate two types of queries: stained query and clean query. A query is stained if the adversary has prior knowledge of its value.

[3]A possible implementation of the trusted component is IBM secure co-processor PCIXCC [1] which connects to a host through a PCI bus.

For example, all PIR queries due to $\mathcal{A}$'s request to $\mathcal{O}$ are stained ones; and an uncompromised user's query is clean. The notion of security is defined as below, similar to the one in ORAM [8]. Namely, no polynomial time adversary gets non-negligible advantage in determining Q by observing access patterns including Q's execution.

Definition 2.1: Let κ be a security parameter. Let Λ_K denote the domain of all access patterns of length K. A private information retrieval scheme is computationally secure if and only if for any PPT $\mathcal{A}$, there exists a PPT $\bar{\mathcal{A}}$, such that for any target clean query Q, $\forall K \in \mathbb{N}, \forall \lambda_K \in \Lambda_K$, $\forall q \in [1, n]$,

$$|\Pr(q \leftarrow \mathcal{A}_Q^{\mathcal{O}}(\lambda_K)) - \Pr(q \leftarrow \bar{\mathcal{A}}_Q^{\mathcal{O}}(1^\kappa))| < \epsilon(\kappa)$$

where $\epsilon(\kappa)$ is a negligible function, and $q \leftarrow \mathcal{A}_Q^{\mathcal{O}}(\lambda)$ and $q \leftarrow \bar{\mathcal{A}}_Q^{\mathcal{O}}(1^\kappa)$ denote the events that $\mathcal{A}$ and $\bar{\mathcal{A}}$ output q as the value of Q with the assistance of $\mathcal{O}$, respectively. The probability is taken over all the internal coin tosses of the PIR scheme, Q, and $\mathcal{A}$ or $\bar{\mathcal{A}}$.

C. Protocol Overview

Recall that our predecessors [10], [9], [19] run as follows. Before a session starts, the database is encrypted and permuted using fresh secrets generated by T. During execution, T retrieves the requested item, say d_i, from the database if d_i is not in the cache; otherwise, a random item is fetched to the cache. When the cache is full, the *entire* database is re-shuffled and re-encrypted for the next session. The objective of database shuffles is to re-mix the touched database entries with the untouched ones, so that future executions appear independent with preceding ones. Due to the full database shuffle, these protocols incur $\mathrm{O}(n)$ computation cost.

SECURITY INTUITION Our proposed scheme is rooted at an insightful observation: the full database shuffle is not indispensable, as long as user queries produce access patterns with the same distribution. Note that it is unnecessary to shuffle white records. A white record does not leak any query information for the following two reasons. First, all records are encrypted and therefore a white record itself does not compromise privacy. Secondly, since it is white, there exists no access pattern involving it. Therefore, the observation that an encrypted record is not touched does not help the adversary to derive any information about (existing) user queries, which is the security goal of private information retrieval.

Based on this observation, we propose a new PIR scheme which has a novel retrieval algorithm and a partial shuffle algorithm. In a high level, our scheme proceeds as follows. Initially, all database entries are labeled **white**. Once a record is fetched, it is labeled **black**. For a query on d_i, T executes a novel *twin retrieval* algorithm: if d_i is in the cache, T randomly fetches a pair of records, black and white respectively; otherwise, it retrieves the needed record and another random record in a different color. When the cache is full, T only shuffles and re-encrypts all black records, which is called a *partial shuffle*. Intuitively, H always spots a black and white pair being retrieved for queries in a session. Moreover, the information collected in one session is rendered obsolete for the succeeding sessions because partial shuffles remove the correlations across sessions.

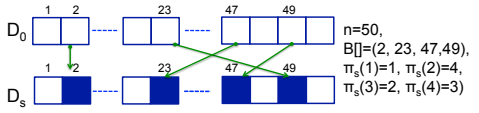


Fig. 1. An illustration of permutation among black records between D_0 and D_s

A challenge of this approach is how T securely decides a record's color and securely retrieves a random record in a desired color. Note that since all accesses to the database appear random, the black records are dispersed across the entire database. It is practically infeasible for an embedded trusted component to "memorize" all state information. A straw-man solution is that T scans the database to check the colors of all records. Nonetheless, this solution is not attractive since its linear complexity totally nullifies our design efforts.

In the next section, we will present a basic PIR scheme by assuming that T's cache is big enough to accommodate the needed auxiliary data structures. In this way, we dismiss the aforementioned challenge for the time being and focus on the new twin retrieval and partial shuffle algorithms and their security. This facilitates an easier presentation of our full scheme in Section IV where we will remove this storage assumption and propose a solution to the mentioned challenge.

III. THE BASIC CONSTRUCTION

A. A Basic PIR Scheme

T manages a sorted array denoted by $B[\ ]$ in its cache. B stores all black addresses in the ascending order. In every session, a constant number of white records are fetched. Therefore, the size of B grows with a constant amount after each session. At the end of the $(s-1)$-th session, T generates $\pi_s : [1, |B|] \rightarrow [1, |B|]$ as the pseudorandom permutation among black addresses for the s-th session. We remark that π_s is not defined upon the entire database. For two black addresses x, y in B, s.t. $B[i] = x, B[j] = y, \pi_s(i) = j$, it implies that $D_s[y]$ contains the same item as in $D_0[x]$. We denote this relation as $\mathsf{D_0[x]} \simeq \mathsf{D_s[y]}$. Namely, π_s specifies the mapping between the black addresses in D_s and their original addresses in D_0. An example of B and π_s is illustrated in Figure 1.

Initialization. T chooses a pseudorandom permutation $\sigma : [1, n] \rightarrow [1, n]$ and an encryption algorithm $\mathcal{E}$ with a random secret key sk_0. It encrypts and shuffles D into D_0 by applying σ and $\mathcal{E}_{sk_0}()$. This step can also be performed by a trusted authority which then initializes T accordingly.

Session 0. T executes $k/2$ queries using the retrieval algorithm in [19]. For $Q = i$, if d_i is in T's cache, T reads $D_0[\sigma[i]]$ into the cache. Otherwise, T retrieves a random record. After $k/2$ executions, B is populated with $k/2$ addresses and T generates a secret permutation $\pi_1 : [1, k/2] \rightarrow [1, k/2]$ and a new secret key sk_1. It shuffles the $k/2$ black records according to π_1 while leaving the white records intact. Since all records to be shuffled are in the cache, T simply re-encrypts them using $\mathcal{E}_{sk_1}()$ and then writes the ciphertexts out in a batch to

generate D_1. It deletes all data items in the cache. Note that no read access to the database is needed during this shuffle.

The relations among databases can now be illustrated as $D \overset{\sigma}{\Longrightarrow} D_0$(black addresses) $\overset{\pi_1}{\longrightarrow} D_1$(black addresses). With B, π_s and σ, the record in D_1 storing d_i can be located. For the convenience of presentation, the tuple (i, d_i) is always stored together. Therefore, T can determine d_i's index i without further look-up. This applies for all subsequent sessions.

Session $s \geq 1$**.** For a user query, T executes the twin-retrieval algorithm shown in Algorithm 1. At the end of the session, it executes the partial-shuffle algorithm shown in Algorithm 2 to permute all black records whose addresses are stored in B.

Algorithm 1 Basic Twin Retrieval Algorithm in Session $s \geq 1$.

INPUT: a query on d_i, B. OUTPUT: d_i.

1: **if** d_i not in the cache **then**
2: $j \leftarrow \sigma(i)$.
3: $u \leftarrow binary\_search(j, B)$;
4: **if** $u \neq NULL$ **then**
5: d_i is black; set $v \leftarrow B[\pi_s(u)]$ and read $D_s[v]$ and read a random white record;
6: **else**
7: d_i is white; read a random black record and read $D[j]$ which stores d_i;
8: **end if**
9: **else**
10: read a random black record and a white record from D_s into the cache.
11: **end if**
12: return d_i to the user.

Remark 1: $binary\_search(j, B)$ is a standard binary search function. If j is B, it returns u, such that $B[u] = j$; otherwise it returns $NULL$.

Remark 2: T generates a random white record by repetitively generating $x \in_R [1, n]$ until $binary\_search(x, B)$ returns $NULL$.

Explanation of Algorithm 1. For a query on d_i , T runs the algorithm to fetch one black record and one white record from D_s. The decryption of the fetched record is performed within T. If the needed item is in the cache, both records are retrieved randomly. Otherwise, it fetches the requested record for d_i and the other one in a different color.

Note that once a record becomes black, it stays in black forever. T searches B to determine the color of the requested record. If it is white, T directly uses its image under σ to read the data, since it has never been shuffled. Otherwise, T uses π_s and looks up B to locate its present address in D_s. After $k/2$ queries, the cache is full, where half entries are black and half are white, meaning that they are newly retrieved. Note that B now has $(1+s)k/2$ entries with $k/2$ newly added ones during the s-th session.

Explanation of Algorithm 2. The partial shuffle is to re-mix up all black records whose addresses are in B. For each partial shuffle, a new permutation is selected so that all black records are assigned to addresses randomly chosen from B. Note that the reshuffle process also re-encrypts them so that the server is unable to link any black record's old address and new address.

Algorithm 2 Basic Partial Shuffle Algorithm executed by T at the end of s-th session, $s > 0$

INPUT: B with $(1 + s)k/2$ black records. OUTPUT: D_{s+1}

1: secretly generate a random permutation $\pi_{s+1} : [1, |B|] \rightarrow [1, |B|]$, and a new key sk_{s+1}.
2: **for** $(I = I_f = 1; I \leq |B| - k; I++)$ **do**
3: /* Increase I_f, until the corresponding item d_t is not in cache. Fetch d_t */
4: **while** TRUE **do**
5: $j \leftarrow \pi_{s+1}^{-1}(I_f)$; $t \leftarrow \sigma^{-1}(B[j])$;
6: **if** d_t in the cache, $I_f \leftarrow I_f + 1$; **else break**;
7: **end while**
8: /* We need to translate the record addresses across different permutations */
9: $\delta \leftarrow |\{d_i | d_i$ is in cache and is white and $\sigma(i) < B[j]\}|$, $v \leftarrow \pi_s(j - \delta)$;
10: $\delta \leftarrow |\{d_i | d_i$ is in cache and is white and $\sigma(i) < B[v]\}|$, $v \leftarrow v + \delta$;
11: fetch $D_s[B[v]]$ as d_t.
12:
13: /* Write to D_{s+1} with either d_t or the right one in the cache*/
14: **if** $I = I_f$ **then**
15: write $\mathcal{E}_{sk_{s+1}}(d_t, t)$ into $D_{s+1}[B[I]]$;
16: **else**
17: Insert (t, d_t) into the cache.
18: $j \leftarrow \pi_{s+1}^{-1}(I)$; $t \leftarrow \sigma^{-1}(B[j])$.
19: Retrieve d_t from the cache and write $\mathcal{E}_{sk_{s+1}}(d_t, t)$ to $D_{s+1}[B[I]]$.
20: **end if**
21: $I_f = I_f + 1$;
22: **end for**
23: encrypt and write the remaining k records in the cache to D_{s+1} accordingly, securely eliminate π_{s-1}. Quit the s-th session.

Essentially, it is to permute those black records in D_s into D_{s+1} according to $\pi_{s+1} : [1, |B|] \rightarrow [1, |B|]$. T fills in D_{s+1} following the address sequence: $B[1], B[2], \ldots$ and so on. The record written into $D_{s+1}[B[i]]$ may be available in T's cache; and if not, it can be found in D_s. In the latter case, the record can be located through an address lookup, which in a high level view is through $D_{s+1} \overset{\pi_{s+1}^{-1}}{\Longrightarrow} D_0 \overset{\pi_s}{\longrightarrow} D_s$. Note that since π_{s+1} and π_s are defined in different domains due to the growth of B, the algorithm has to make necessary address adjustment as in Step 13 and 14. The adjustment is done by calculating δ which is the number of white items in the cache making the difference.

In order to prevent the server from knowing whether the record to write is from the cache or fetched from D_s, T always retrieves one record from D_s. Therefore, if the record is from the cache, T finds the next black address ($B[i']$) whose pre-

image under π_{s+1} is not in the cache (as shown in the **while** loop) and fetches it into the cache.

B. Security Analysis

Our analysis is to show that the PIR scheme presented above satisfies Definition 2.1. Intuitively, we show that the adversary gets negligible advantage by obtaining the transaction scripts which is computationally indistinguishable from a random string of the same length. We begin with a definition of *ideal implementation* which dismisses those attacks on the permutations and encryptions.

Definition 3.1 (Ideal Implementation): An implementation of Algorithm 1 and 2 is said to be an ideal implementation if all permutations $\sigma, \pi_1, \pi_2, \cdots$ are true random permutations on their respective domains and $\mathcal{E}$ is information theoretically secure. ■

Let $\mathtt{A}_i$ be the random variable denoting the script recording all accessed database addresses for retrieval during the i-th session, and let $\mathtt{R}_i$ be the random variable denoting the script recording all accessed database addresses for partial shuffle at the end of the i-th session. Lemma 3.1 shows that the partial shuffle in an ideal implementation is uniform, in the sense that after the partial shuffle, all black records are uniformly remixed. Thus, all black records appear indistinguishable to H. Then, Lemma 3.2 proves that at any time, the access patterns for all query executions have the identical distribution. Finally, we prove in Theorem 3.1 that the basic PIR scheme (without an ideal implementation) satisfies Definition 2.1.

Lemma 3.1 (Uniform Shuffle): For an ideal implementation, Algorithm 2 at the end of the s-th session performs a uniform shuffle on all $(1+s)k/2$ black records. Namely, $\forall s \geq 0$, $\forall i, j \in [1, (1+s)k/2]$, $\forall \mathtt{A}_t, \mathtt{R}_t, 0 \leq t < s$,

$$\Pr(D_{s+1}[B[j]] \simeq D_0[B[i]] \mid \mathtt{A}_0, \mathtt{R}_0, \mathtt{A}_1, \cdots, \mathtt{A}_{s-1}, \mathtt{R}_{s-1}) = \frac{2}{(1+s)k} \quad (1)$$

where the probability is over all the random coin tosses in generating permutation $\sigma, \pi_1, \cdots, \pi_{s-1}$ and in query executions. ■

PROOF. We prove the lemma by an induction on the session number.

I. $s = 0$. In the end of the session 0, all $k/2$ black records are in the cache and are written out to D_1 directly from the cache. Thus, no matter what π_1 is in use, $\mathtt{R}_0$ remains the same. Therefore, when π_1 is a true random permutation, the probability $\Pr(D_1[B[j]] \simeq D_0[B[i]] \mid \mathtt{A}_0, \mathtt{R}_0) = 2/k$ holds for all $1 \leq i, j \leq k/2$.

II. Suppose that Equation 1 holds for $s = m-1$, i.e.

$$\Pr(D_m[B[j]] \simeq D_0[B[i]] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_{m-1}, \mathtt{R}_{m-1}) = 2/mk$$

after the $(m-1)$-th session. We now consider the m-th session. Let W denote the set of $k/2$ white addresses touched and turned into black during the m-th session. We use B' to denote the new version of the black address array at the end of the m-th session. To simplify the presentation, let $B' = [b'_1, b'_2, \cdots, b'_{(1+m)k/2}]$, $B = [b_1, b_2, \cdots, b_{mk/2}]$ and $W = [w_1, w_2, \cdots, w_{k/2}]$. Note $B' = B \cup W$. We proceed to prove that Equation 1 also holds for $s = m$, i.e.,

$$\Pr(D_{m+1}[b'_j] \simeq D_0[b'_i] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_m, \mathtt{R}_m) = 2/(1+m)k,$$

for all $i, j \in [1, (1+m)k/2]$.

For the ease of presentation, we use Φ to denote $\Pr(D_{m+1}[b'_j] \simeq D_0[b'_i] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_m, \mathtt{R}_m)$. Further, we define p_x and q_x as $p_x \triangleq \Pr(D_{m+1}[b'_j] \simeq D_m[x] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_m, \mathtt{R}_m)$ and $q_x \triangleq \Pr(D_m[x] \simeq D_0[b'_i] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_m, \mathtt{R}_m)$. Since π_{m+1} and π_m are true random permutations, $\mathtt{R}_m$ has no effect in determining whether $D_m[x] \simeq D_0[b'_i]$. Thus, $q_x = \Pr(D_m[x] \simeq D_0[b'_i] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_m)$. Since D_{m+1} is generated based on D_m, we have $\Phi = \sum_{x \in B'} p_x \cdot q_x$. We will evaluate Φ with this formula by distinguishing $D_m[x]$ in the cache and $D_m[x]$ not in the cache.

Define $U = \{x | x \in B', D_m[x] \text{ in cache}\}$ and $V = B' \setminus U$. Note that $|U| = k$ and $|V| = (m-1)k/2$. Thereafter,

$$\Phi = \sum_{x \in U} p_x q_x + \sum_{x \in V} p_x q_x$$

For $x \in U$, the adversary obtains no information about $D_m[x]$'s new address in D_{m+1} as it is written out directly from the cache, which is independent from the access patterns. Therefore, for all $x \in U$, $p_x = 2/(1+m)k$. We have

$$\Phi = \frac{2}{(1+m)k} \sum_{x \in U} q_x + \sum_{x \in V} p_x q_x \quad (2)$$

In addition, since either $x \in U$ or $x \in V$, we have

$$\sum_{x \in V} p_x = 1 - \sum_{x \in X} p_x = 1 - 2/(m+1) = (m-1)/(m+1)$$

We evaluate Φ using Equation 2 in two exclusive cases. I) $D_0[b'_i]$ corresponds to a white record retrieved during the m-th session; II) $D_0[b'_i]$ corresponds to a black record retrieved during the m-th session.

Case I. $D_0[b'_i]$ **is white**. Since $D_0[b'_i]$ is white until the m-th session, its corresponding record at D_m must be in the cache. Therefore, for $\forall x \in V$, $q_x = 0$, and $\sum_{x \in U} q_x = 1$. Thus, we have

$$\Phi = \frac{2}{(1+m)k} \sum_{x \in X} q_x + \sum_{x \in Y} p_x q_x = \frac{2}{(1+m)k}$$

Case II. $D_0[b'_i]$ **is black** Note that there are attacks whereby $\mathtt{A}_m$ allows the adversary to know that for some $\bar{x}$, $D_m[\bar{x}] = D_0[b'_i]$. We consider two subcases below.

Case II.A. The adversary knows $D_0[b'_i] = D_m[\bar{x}]$ by issuing tainted queries. Therefore, for $x \in V$, $q_x = 0$ since $D_m[x]$ must be in the cache. In addition, for $x \in U$ and $x \neq \bar{x}$, $q_x = 0$ as only $\bar{x}$ is the matching one. Therefore, since $q_{\bar{x}} = 1$, we have

$$\Phi = p_{\bar{x}} q_{\bar{x}} = 2/(1+m)k$$

Case II.B. The adversary does not know any $\bar{x}$ satisfying $D_0[b'_i] = D_m[\bar{x}]$. (Note that the adversary may still know $D_m[x]$ using a tainted query.)

If $x \in V$, $D_m[x]$ is not involved in the m-th session. It means that $(\mathtt{A}_m, \mathtt{R}_m)$ do not affect the probability $D_m[x] = D_0[b'_i]$. Therefore, by the induction assumption, for $x \in V, q_x = \frac{2}{mk}$.

If $x \in U$, $D_m[x]$ is involved in one of the queries executed in the m-th session. For those $x \in U$ and $D_m[x]$ is white before the m-th session, we have $q_x = 0$ because $D_0[b'_i]$ is black. For those $x \in U$ and $D_m[x]$ is black, because $\mathcal{E}()$ is information theoretically secure, $\mathtt{A}_m$ does not give the adversary any advantage to determine whether $D_m[x] = D_0[b'_i]$. Therefore, $q_x = \Pr(D_m[B[j]] \simeq D_0[B[i]] \mid \mathtt{A}_0, \mathtt{R}_0, \cdots, \mathtt{A}_{m-1}, \mathtt{R}_{m-1})$. By the induction assumption on $s = m-1$, we have $q_x = 2/mk$ if $x \in U$ and x is black in Case II.B. Note that there are exactly $k/2$ such black x-s Therefore, we have

$$\Phi = \sum_{x \in U, x \text{ is black}} p_x q_x + \sum_{x \in V} p_x q_x = \frac{2}{(1+m)k} \cdot \frac{2}{mk} \cdot \frac{k}{2} + \frac{2}{mk} \cdot \frac{m-1}{m+1} = 2/(1+m)k$$

Combining all the cases above, we have proved that when $s = m$, all $i, j \in [1, (1+m)k/2]$, $\Phi = 2/(1+m)k$. With **I** and **II**, we conclude the proof of the lemma. □

Lemma 3.2 (Uniform Access): Let Q be the random variable for a clean query. Let (X, Y) be the two-dimensional random variable for the black and white addresses accessed in the twin retrieval algorithm for Q. Let $\mathtt{A}$ denote the access pattern when Q is executed. Then, in an ideal implementation, $\forall q_1, q_2 \in [1, n]$, $\Pr(X = x, Y = y \mid \mathtt{A}, Q = q_1) = \Pr(X = x, Y = y \mid \mathtt{A}, Q = q_2)$.■

PROOF. Without loss of generality, suppose that Q is executed at the s-th session. 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 based on σ. 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 Algorithm 1, for a query Q, a black record and a white record are read. Let I_0 denote the set of indexes of all items in the cache, and I_1 denote the set of indexes whose corresponding records are black records and not presently in the cache, and $I_2 = [1, n] \setminus (I_0 \cup I_1)$, which is the set of indexes whose corresponding records are white. To prove the theorem, it is sufficient to demonstrate that for any $q \in [1, n]$, $\Pr((X = x, Y = y) \mid \mathtt{A}, Q = q)$ remains the same in the following cases covering all possibilities of q.

- Case (0) $q \in I_0$. Both X and Y are randomly retrieved. So $\Pr((X = x, Y = y) \mid \mathtt{A}, Q = q) = (\frac{1}{|B|}, \frac{1}{n-|B|})$
- Case (1) $q \in I_1$. T reads the corresponding black record and a random white record from D_s. Due to Lemma 3.1, the corresponding record could be in any position in B with the same probability. Therefore $\Pr(X = x \mid \mathtt{A}, q) = \frac{1}{|B|}$. Y is a random retrieval, which is independent of $\mathtt{A}$. Therefore, $\Pr((X = x, Y = y) \mid \mathtt{A}, Q = q) = (\frac{1}{|B|}, \frac{1}{n-|B|})$.
- Case (2) $q \in I_2$. 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, $\Pr(Y = y \mid \mathtt{A}, q) = \frac{1}{n-|B|}$. X is a random retrieval independent from $\mathtt{A}$. Therefore $\Pr((X = x, Y = y) \mid \mathtt{A}, Q = q) = (\frac{1}{|B|}, \frac{1}{n-|B|})$.

By combining all the cases above, we complete the proof. □

Note that the ideal implementation is infeasible in practice. Given the limited resource T has, all permutations are pseudo-random rather than true random, and $\mathcal{E}()$ is semantically secure only against a PPT adversary. Therefore, we prove below that under a practical implementation where $\sigma, \pi_0, \pi_1, \cdots$ and $\mathcal{E}$ are computationally secure, our PIR scheme consisting of Algorithm 1 and 2 is computationally secure. The proof is based on a series of games [16] between an adversary $\mathcal{A}$ and a challenger who simulates the setting with respect to our scheme.

Theorem 3.1 (Main Theorem): The proposed PIR scheme satisfies Definition 2.1. Namely, for a database $D = [d_1, d_2, \cdots, d_n]$ and any D_s using pseudo-random permutations $\sigma, \pi_1, \cdots, \pi_s$ and a semantically secure encryption $\mathcal{E}$, for any PPT adversary $\mathcal{A}^{\mathcal{O}}$ against our scheme (Algorithm 1 and 2), there exists a PPT $\bar{\mathcal{A}}^{\mathcal{O}}$, such that for any target clean query Q, for any $K \in \mathbb{N}$, for any access pattern λ_K, $\forall q \in [1, n]$,

$$|\Pr(q \leftarrow \mathcal{A}_Q^{\mathcal{O}}(\lambda_K)) - \Pr(q \leftarrow \bar{\mathcal{A}}_Q^{\mathcal{O}}(1^\kappa))| < \epsilon(\kappa)$$

where $\epsilon(\kappa)$ is a negligible function. The probability is taken over all the internal coin tosses of Algorithm 1, Algorithm 2, Q, and $\mathcal{A}$ or $\bar{\mathcal{A}}$.

PROOF. In the proof, we do not differentiate access patterns for query execution or for partial shuffle.

Game 0. Game 0 is defined as the attack game between $\mathcal{A}$ and the challenger. Game 0 is conceptually equivalent to $\mathcal{A}$' attack against our scheme without an ideal implementation. In particular, the challengers sets up the system and simulates T using pseudo-random permutations $\sigma, \pi_1, \pi_2, \cdots$, and a semantically secure encryption scheme $\mathcal{E}$. $\mathcal{A}$ queries $\mathcal{O}$ to run tainted queries. $\mathcal{A}$ observes all access patterns for all queries.

At certain point, $\mathcal{A}$ asks for a challenging query denoted Q. The challenger randomly sets $Q = q$, where $q \in_R [1, n]$, and executes the query. $\mathcal{A}$ observes its execution. Then $\mathcal{A}$ continues to query $\mathcal{O}$ and to observe the access pattern. After totally observing a polynomial number (K) of database accesses (λ_K), $\mathcal{A}$ halts and outputs $q' \in [1, n]$. $\mathcal{A}$ wins Game 0 if $q' = q$. We define s_0 to be the event that $q' = q$. Note that $\Pr(s_0) = \Pr(q \leftarrow \mathcal{A}_Q^{\mathcal{O}}(\lambda_K))$.

Game 1. We transform Game 0 into Game 1 with the following changes. Instead of using pseudo-random permutations $\sigma, \pi_1, \cdots$, the challenger uses true random permutations $\sigma', \pi'_1, \cdots$, which are on the same respective domains. All others remain the same as in Game 0.

Let s_1 be the event that $q' = q$ in Game 1. We claim that $|\Pr(s_1) - \Pr(s_0)| = \epsilon_p$, where ϵ_p is the advantage for any PPT adversary in distinguishing between a pseudo-random permutation and a random permutation over the same permutation domain. This can be easily proven by using a distinguisher attacking the pseudo-random permutation generator by interpolating between Game 0 and Game 1. To determine whether a family of permutations are true random or not, the distinguisher applies them in Game 0 and Game 1 and observes $\mathcal{A}$'s success probabilities in two games.

Game 2 We transform Game 1 into Game 2 with the following changes. Instead of using a semantically secure encryption $\mathcal{E}$, the challenger uses an information theoretically secure

encryption algorithm $\mathcal{E}'$. All others remain the same as in Game 1.

Let s_2 be the event that $q' = q$ in Game 2. We claim that $|\Pr(s_2) - \Pr(s_1)| = \epsilon_e$, where ϵ_e is the advantage of a PPT adversary in distinguishing $\mathcal{E}$ and $\mathcal{E}'$, i.e. breaking the semantic security of $\mathcal{E}$. Similar to the proof in Game 1, our claim here can be proven by constructing a distinguisher which is given oracle accesses to either $\mathcal{E}$ or $\mathcal{E}'$ and interpolates between Game 1 and Game 2.

Therefore, we have $|\Pr(s_2) - \Pr(s_0)| = \epsilon_p + \epsilon_e$. Thus, the theorem can be proved by proving $\Pr(s_2) = \Pr(q \leftarrow \bar{\mathcal{A}}_Q^{\mathcal{O}}(1^\kappa))$, which is equivalent to prove that in an ideal implementation of our proposed PIR scheme, $\forall \lambda_K, \Pr(Q = q \mid \lambda_K) = \Pr(Q = q)$. By the conditional probability formula, it is equivalent to prove that $\forall \lambda_K$, $\Pr(\lambda_K \mid Q = q) = \Pr(\lambda_K)$. Fix any session s, we prove it by an induction on K.

I: When $K = 1$, our target is to prove that $\Pr(X = x, Y = y \mid Q = q) = \Pr(X = x, Y = y)$, $\forall x, y \in [n]$. Note that $\Pr(x, y) = \sum_{i=1}^{n} \Pr(x, y \mid i)\Pr(i)$. Consider $\Pr(x, y \mid i)$. There are two cases:

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

Thus, in both case $\Pr(x, y \mid i) = (\frac{1}{|B|}, \frac{1}{n-|B|})$ 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: (**induction assumption**) For $K' = K - 1, \Pr(\lambda_{K'} \mid Q = q) = \Pr(\lambda_{K'})$. We then prove that $\Pr(\lambda_K \mid Q = q) = \Pr(\lambda_K)$. Without loss of generality, let $\lambda_K = \lambda_{K'} \cup (x, y)$, where $(X = x, Y = y)$ is the K-th database read. Therefore, to prove $\Pr(\lambda_{K'}, (x, y) \mid q) = \Pr(\lambda_K)$, it is sufficient to prove $\Pr(x, y \mid \lambda_{K'}, q) = \frac{\Pr(\lambda_{K'}, x, y)}{\Pr(\lambda_{K'} \mid q)}$ due to the conditional probability formula. By the induction assumption, it is sufficient to prove $\Pr(x, y \mid \lambda_{K'}, q) = \Pr(x, y \mid \lambda_{K'})$. We prove this by considering 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, $\lambda_{K'}$ and (x, y) are independent of $Q = q$. So, $\Pr(x, y \mid \lambda_{K'}, q) = \Pr(x, y \mid \lambda_{K'})$.

CASE 2: Note that $\Pr(x, y \mid \lambda_{K'}) = \sum_{q=1}^{n} \Pr(x, y \mid \lambda_{K'}, q)\Pr(q \mid \lambda_{K'})$, where $Q = q$ is the query corresponding (x, y). Due to Lemma 3.2, $\Pr(x, y \mid \lambda_{K'}, q) = \Pr(x, y \mid \lambda_{K'}, q')$, $\forall q, q' \in [1, n]$. Therefore, $\Pr(x, y \mid \lambda_{K'}) = \Pr(x, y \mid \lambda_{K'}, q) \sum_{i=1}^{n} \Pr(i \mid \lambda_{K'})$. According to the induction, $\Pr(i \mid \lambda_{K'}) = \Pr(i)$, we have $\Pr(x, y \mid \lambda_{K'}) = (x, y \mid \lambda_{K'}, q)$

CASE 3: Let Q' be the random variable for the query which generates (x, y). Considering all possible values of Q', denoted by q', we have $\Pr(x, y | \lambda_{K'}, q) = \sum_{q'=1}^{n} \Pr(x, y \mid \lambda_{K'}, q, q')\Pr(q' \mid \lambda_{K'}, q)$. Note that $\Pr(x, y \mid \lambda_{K'}, q, q') = \Pr(x, y \mid \lambda_{K'}, q')$ since (x, y) is determined by Q' and $\lambda_{K'}$ according to our PIR algorithm. Therefore, $\Pr(x, y | \lambda_{K'}, q) = \sum_{q'=1}^{n} \Pr(x, y \mid \lambda_{K'}, q')\Pr(q' \mid \lambda_{K'}, q)$. Since $Q' = q'$ is independent of $\lambda_{K'}$ and $Q = q$, thus

$$\Pr(x, y | \lambda_{K'}, q) = \sum_{q'=1}^{n} \Pr(x, y \mid \lambda_{K'}, q')\Pr(q' \mid \lambda_{K'}) = \Pr(x, y \mid \lambda_{K'})$$

Hence, for all three cases, $\Pr(x, y \mid \lambda_{K'}, q) = \Pr(x, y \mid \lambda_{K'})$, which proves that $\Pr(Q = q \mid \lambda_K) = \Pr(Q = q)$. Thus, we conclude with $|\Pr(s_2) - \Pr(s_0)| = \epsilon_p + \epsilon_e$. □

IV. A CONSTRUCTION WITHOUT STORAGE ASSUMPTION

In this section, we consider the scenario that T does not have the capability for storing B whose size grows linearly to the number of queries. B is therefore maintained by H. Note that unprotected accesses to B may leak information about the black records T looks for, and consequently compromise query privacy. A straightforward solution is to treat B as a database, and to run another PIR query on it. Nonetheless, the cost of this nested PIR approach seriously counteracts our efforts to improve the computational efficiency.

We devise two tree structures denoted by Γ and Ψ stored in H to facilitate T's accesses on black and white records respectively. We also retrofit the previous twin-retrieval and partial-shuffle algorithms such that the accesses to Γ and Ψ are oblivious, since all accesses to Γ and Ψ appear uniformly to H for all query executions.

A. Auxiliary Data Structures

Here we only describe the data structures and the involved algorithms. Their construction and the security analysis are presented in the subsequent sections.

1) Management of Black Records: H maintains two arrays: B and $\hat{B}$, recording black addresses as in Section III. The latter array is for T to acquire session related information. When a session starts, B and $\hat{B}$ are identical. During the session, only B is updated with every database access as in the previous scheme, and $\hat{B}$ is not updated. In the beginning of a session, H overwrites $\hat{B}$ with B which has $k/2$ more elements.

2) Management of Permutation: Recall that D_s is a result of a partial shuffle under the permutation $\pi_s : [1, |B|] \rightarrow [1, |B|]$. The permutation can essentially be represented by $|B|$ pairs of tuples (x, y), where $x \in [1, n]$ is the item's index in D and $y \in [1, |B|]$ is the corresponding record's address in B. T selects a cryptographic hash function $H()$ with a secret key v and a CPA secure symmetric key encryption scheme with a secret key u, where the encryption and decryption functions are denoted by $\mathcal{E}_u()$ and $\mathcal{D}_u()$ respectively. We use $\overline{x}$ to denote $\mathcal{E}_u(H(x, v))$. Therefore, the permutation can be represented by a 2-tuple list $L = [(\overline{x_1}, y_1), (\overline{x_2}, y_2) \cdots, (\overline{x_{|B|}}, y_{|B|})]$, sorted under $H(x, v)$ values, i.e. $H(x_1, v) < \cdots < H(x_{|B|}, v)$. Let Γ be a *complete binary search tree* with $|B| - 1$ randomly assigned inner node and with L being the $|B|$ leaves such that an inner node stores $\mathcal{E}_u(a)$ satisfying that a is larger than the plaintext stored in its left child and smaller than the plaintext of its right child. Hereafter, we refer to the plaintext stored in L and Γ as *keys* as they are used for numerical comparison. Figure 2 depicts one toy example of Γ with 4 leaves.

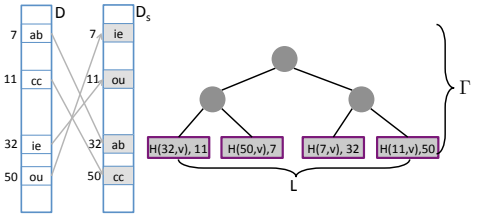


Fig. 2. An illustration of Γ, where the black address array $B = [7, 11, 32, 50]$ and the permutation can be represented as $(7, 32), (11, 50), (32, 11), (50, 7)$. $L = \{(E_u(H(32, v)), 11), (E_u(H(50, v)), 7), (E_u(H(7, v)), 32), (E_u(H(11, v)), 50)\}$, and $H(32, v) < H(50, v) < H(7, v) < H(11, v)$. The shadows in Γ implies that all nodes are encrypted.

We design three algorithms on Γ: **random-walk**, **targeted-walk**(x) and **secure-insert** as described below.

- **random-walk**: Starting from the root of Γ, T fetches a node from Γ into the cache, and secretly tosses a random coin such that its both child nodes have the same probability to be fetched in the next level. The process is repeated until a leaf node is fetched.
- **targeted-walk**(x): Starting from the root of Γ, T fetches a node from Γ into its cache and gets its key by decryption. If x is less than or equal to the key, T fetches its left child; otherwise fetches the right child. The process is repeated until a leaf node is reached.
- **secure-insert**(a, b, L) where L has been sorted under $H()$ values: The same as the regular insertion algorithm to a sorted list, except that all comparisons are performed within T's cache after decryption, and that $(\mathcal{E}_u(a), b)$ are inserted into L instead of (a, b) in plaintext.

The **random-walk** algorithm implements fetching a random black record, whereas the **targeted-walk** algorithm performs a real binary search. Both algorithms walk from the root of Γ downwards to a leaf node, i.e. an entry in L. These two algorithms are used during query execution whereas **secure-insert** is used in constructing L.

3) Management of White Addresses: We need to manage those white records as well. The black addresses virtually divide $[1, n]$ into white segments, i.e. blocks of adjacent white addresses. We use an array denoted by Ψ to represent the white segments. An entry in Ψ has three fields, i.e. $\Psi[i] = \langle l, m, M \rangle$ representing the i-th white segment which starts from the address $\Psi[i].l$ and contains $\Psi[i].m$ entries with l being the M-th white address in the database. Namely, $\Psi[i].M = \sum_{j=1}^{i-1} \Psi[j].m + 1$. Since Ψ does not hold any secret information, it is managed and stored by H. Nonetheless, similar to the security requirement of Γ, the notion of PIR also requires that the server cannot distinguish whether T's access to Ψ is for a random white record or one requested by a query. T utilizes Ψ to fetch white records in the following two ways.

- Random-search: T generates $r \in_R [1, n - |B|]$. Then it runs a binary search on Ψ for the r-th white record in D_s, which stops at $\Psi[i]$ satisfying $\Psi[i].M \leq r < \Psi[i+1].M$. It computes $y = \Psi[i].l + r - \Psi[i].M$ and fetches the y-th record from D_s.
- Targeted-search: T runs a targeted search for given an index x whose corresponding address is white. T runs a binary search on Ψ for the address $\sigma(x)$. The search stops at $\Psi[i]$ satisfying $\Psi[i].l \leq \sigma(x) < \Psi[i+1].l$. Then, T fetches the $\sigma(x)$-th record from D_s. Note that the only purpose of this search is to prevent the adversary from distinguishing whether a white record is randomly selected or not.

a) Management of Cache: We need to store more information in the cache. First, we define the term **BIndex** for black records. For a black address x, its BIndex is i iff $B[i] = x$, namely its rank B. The cache is organized as a table denoted by $\mathcal{C}$ whose format is shown in Table II.

$\mathcal{C}$	BIndex	Color	From	Ind	Data
i	5	W	100	2	0x1200
...	...	...	...	...	...

TABLE II
AN EXAMPLE OF $\mathcal{C}$. $\mathcal{C}[i]$ IS $D_s[100]$, WHICH IS A CIPHERTEXT OF (x, d_x) WHERE $x = 2, d_x = 0x1200$. $D_s[100]$ IS WHITE BEFORE BEING RETRIEVED, AND IT BECOMES THE 5-TH BLACK RECORD IN D_s.

The entries in $\mathcal{C}$ are sorted under their *From* fields. Suppose that T fetches a record $D_s[y]$ storing (x, d_x). It inserts a new entry $\mathcal{C}[i]$ into $\mathcal{C}$, where the $\mathcal{C}[i]$.From $= y$, $\mathcal{C}[i]$.Ind $= x$ and $\mathcal{C}[i]$.Data $= d_x$; $\mathcal{C}[i]$.Color is set to 'B' if $D_s[y]$ was black; otherwise $\mathcal{C}[i]$.Color is set to 'W'. In our example shown in Table II, d_i's image in D_0 is currently the 5-th black record in the database.

B. The Scheme

We are now ready to present the full scheme without the assumption of T's storage for B. The scheme consists of Algorithm 3 for query executions and Algorithm 4 for the partial shuffle. In the high level, these two algorithms shares the same logic as Algorithm 1 and 2 in Section III-A. The differences are mainly on how to locate the black and white records needed by protocol execution and how to construct Γ.

1) Twin Retrieval: The main challenge of the retrieval algorithm is to obliviously and efficiently determine a queried record's color and to generate the proper random address. The basic idea of Algorithm 3 is to utilize Γ to determine a record's color by searching for the corresponding ciphertext. If it is black, the search process outputs its exact location; otherwise, it outputs a random black record. To ensure that a leaf can only be retrieved once, T stores the intervals for those retrieved leaves into a temporary set $\mathcal{X}$. For a binary search tree, each leaf has a corresponding interval determined by the two adjacent inner nodes in an in-order traversal. Thus, whether a leaf is retrieved depends on whether the searching key falls into the leaf's interval. Note that these two inner nodes are on the path from the root to the leaf. If the leaf is its parent's left/right child, the recorded ancestor is the nearest one such that the leaf's is is right/left offspring.

More specifically, T differentiates three scenarios. In the first scenario (Line 2), the queried record is in the cache. In this case, it fetches a random black and white pair. In the second scenario (Line 8), the queried record is not in the

cache and the expected search path has been walked priorly, which indicates that the queried record is white. Therefore, T performs a random walk in Γ to avoid repeating the path. In the last case (Line 11), T performs a targeted walk to search for ciphertext c. There are two possible outcomes: the leaf node matches c, indicating that the desired record is black; or the leaf does not match, indicating that the record is white.

Algorithm 3 The General Twin Retrieval Algorithm in Session $s \geq 1$, executed by T.

INPUT: a query on d_i, B, key u, set $\mathcal{X}$, Γ whose root is α. OUTPUT: d_i.

1: $i' \leftarrow \sigma(i)$;
2: **if** $i \in \mathcal{C}$.Ind **then**
3: /* the data item is in the cache */
4: $\langle \overline{x}, y \rangle \leftarrow$**random-walk**; $j_b \leftarrow y$.
5: random-search Ψ which returns j_w, go to Line 17;
6: **end if**
7: $c \leftarrow H(i, u)$;
8: **if** $\exists (l, r, \overline{x'}, y') \in \mathcal{X}$ satisfying $l \leq c < r$ **then**
9: $\langle \overline{x}, y \rangle \leftarrow$**random-walk**; $j_b \leftarrow y$; goto Line 15;
10: **end if**
11: $\langle \overline{x}, y \rangle \leftarrow$**targeted-walk**$(c)$; $j_b \leftarrow y$;
12: **if** $\mathcal{D}_u(\overline{x}) = c$ **then**
13: random-search Ψ which returns j_w. /*the queried record is black.*/
14: **else**
15: targeted-search Ψ for i'. Then $j_w \leftarrow i'$. /*the queried record is white.*/
16: **end if**
17: $\mathcal{X} \leftarrow \mathcal{X} \cup \{(l, r, \overline{x}, y)\}$ where l, r are the the plaintext of leaf $(\overline{x}, y)$'s parent node and one of its ancestors on the path and $l < r$.
18: read the $D_s[j_b]$ and $D_s[j_w]$. After decryption, create two new $\mathcal{C}$ entries for them accordingly. Note that the BIndex is empty for the time being.
19: return d_i to the user.

2) Partial Shuffle: The partial shuffle algorithm shown in Algorithm 4 is the same as Algorithm 2 with two main differences. First, T uses $\mathcal{C}$ to look for a suitable black record to shuffle out (Line 7), rather than repetitively visiting B. Therefore, for every write to the new database, T only has one access for B and one for the old database (Line 11). Secondly, this algorithm has to construct L' and Γ'. When populating the black entries in the new database (Line 15 and 20), T secure inserts the mapping relation (x, y) into L. Note that it is $\mathcal{E}_{u'}(H(x, v'))$ which is inserted into sorted L'. The concurrence of constructing L and filling the new database does not leak information, since y-values of L are exactly the addresses in array B.

The construction of Γ' is also straightforward. Since Γ' is built as a complete binary search tree with L' being the leaves, its topology of Γ' is calculated when L' is ready. Thus, T can scan L' and build Γ': between two adjacent L' nodes, randomly picks a in the domain of $H()$ and builds an inner node storing $\mathcal{E}_{u'}(a)$. Then, based on the computed tree topology, T sets the pointers for its two children, which could be inner nodes or leaf nodes.

Algorithm 4 Partial Shuffle Algorithm executed by T at the end of s-th session, $s \geq 1$

Input: $\mathcal{C}, B$ **Output:** D_{s+1}, Γ' and L'

1: scan B and assign BIndex for each entry in $\mathcal{C}$. Specifically, for every $1 \leq b \leq |B|$, if $\exists x \in [1, k]$, s.t. $\mathcal{C}[x].From = B[b]$, then set $\mathcal{C}[x].BIndex = b$.
2: generate a secret random permutation $\pi_{s+1} : [1, |B|] \rightarrow [1, |B|]$, and a new encryption key u' and hash key v'.
3: **for** $(I = I_f = 1; I \leq |B| - k; I++)$ **do**
4: $j \leftarrow \pi_{s+1}^{-1}(I_f)$;
5: /* Increase I_f, until the corresponding item d_t is not in cache. Then, fetch d_t from D_s. */
6: **while** TRUE **do**
7: **if** $j \in \mathcal{C}.BIndex$, $I_f \leftarrow I_f + 1$; **else break**;
8: **end while**
9: /* We need to translate the record addresses across different permutations */
10: $\delta \leftarrow |\{x \mid \mathcal{C}[x].Color = W \text{ and } \mathcal{C}[x].BIndex < j\}|$, $v \leftarrow \pi_s(j - \delta)$;
11: fetch $\hat{B}[v]$, and then fetch $D_s[\hat{B}[v]]$ as (t, d_t).
12: /* Write to D_{s+1} and update L' */
13: **if** $I = I_f$ **then**
14: Re-encrypt (t, d_t) into $D_{s+1}[B[I]]$;
15: **secure-insert**$(H(t, v'), B[I], L')$;
16: **else**
17: insert a 4-tuple $\langle 0, \text{'B'}, \hat{B}[v], x, d_x \rangle$ into $\mathcal{C}$.
18: find $l \in [1, k]$ satisfying $\mathcal{C}[l].BIndex = \pi_{s+1}^{-1}(I)$.
19: Re-encrypt $(\mathcal{C}[l].Ind, \mathcal{C}[l].Data)$ and insert the result to $D_{s+1}[B[I]]$.
20: **secure-insert**$(H(\mathcal{C}[l].Ind), v'), B[I], L')$;
21: **end if**
22: $I_f = I_f + 1$
23: **end for**
24: write the remaining k records in the cache to D_{s+1} and assign L' accordingly; securely discard π_s, u.
25: scan L' and construct Γ' based on L'.

C. Security Analysis

Our security analysis below focuses on the new security issues caused by using the auxiliary data structures. First, we prove that the adversary's observation on a path of **targeted-walk** does not reveal any information about the query. Note that for a binary search tree, a leaf node exactly represents one search path. We describe the obliviousness of **targeted-walk** using the following lemma.

Lemma 4.1: For any given L and Γ constructed as above, for a search path ending at $L[i]$ resulting from **targeted-walk** on a query X, $\Pr(X = x_1 | L[i]) = \Pr(X = x_2 | L[i])$ for all $x_1, x_2 \in [1, n]$ and $i \in [1, |B|]$.

PROOF. To prove the lemma, it is sufficient to prove $\Pr(X = x_1, L[i]) = \Pr(X = x_2, L[i])$. Without loss of generality, suppose that an in-order traverse of Γ shows to the adversary that

$$\mathcal{D}_u(L[1]) < \cdots < \mathcal{D}_u(c_{i-1}) < \mathcal{D}_u(L[i]) < \mathcal{D}_u(c_i) < \cdots < \mathcal{D}_u(L[|B|])$$

Therefore, it appears to the adversary that $\Pr(X, L[i])$ is exactly the probability that $\mathcal{D}_u(c_{i-1}) \leq H(X, v) < \mathcal{D}_u(c_i)$, which depends on the hash function $H(\cdot, v)$ and the decryption

function $\mathcal{D}_u(\cdot)$. Under the random oracle model, $H(X, v)$ is uniformly distributed in H's domain. Therefore, $\Pr(X, L[i])$ is determined by $\delta_i = \mathcal{D}_u(c_i) - \mathcal{D}_u(c_{i-1})$. Since $\{\delta_i\}_{i=1}^{|B|}$ are uniformly distributed and the adversary has no information about $\{\delta_i\}_{i=1}^{|B|}$ as every δ_i is only chosen once in query execution, $\Pr(X = x_1|L[i]) = \Pr(X = x_2|L[i]) = 1/|B|$ for all $x_1, x_2 \in [1, n]$. □

Lemma 4.2: Suppose that σ is a true-random permutation over $[1, n]$. For any given Ψ, for all white item d_x, the search path generated by the targeted search for d_x has the same distribution as a random search, where the probability is computed over the coin tosses of σ and in random search.

PROOF. The proof is very straightforward. Since σ is a true random permutation, $\sigma(x)$ is a random position among all white records. Thus, $\sigma(x)$ has the same distribution as $r \in [1, n - |B|]$ which is just the rank of a random white record. Therefore, the two binary searches above result in the identical distribution. □

The two lemmas above show that the independent executions of those component algorithms for auxiliary data structures do not leak information to the adversary. Next, we prove that Algorithm 3 and Algorithm 4 remain as oblivious though auxiliary data structures are accessed within their executions. In plain words, we show that an adversary does not observe more useful information in executing Algorithm 3 and 4 than in executing Algorithm 1 and 2.

Lemma 4.3: The accesses to auxiliary data structures B, Ψ, Γ in Algorithm 3 are oblivious.

PROOF. For all queries, only $B[1]$ and $B[(s + 1)k/2]$ are accessed. In addition, the black addresses stored in B are known to the adversary. Thus, accesses to B do not give the adversary any advantages.

By Lemma 4.2, we have shown that for all queries, the accesses to Ψ in executing Algorithm 3 have the identical distribution. Furthermore, the trace of searching on Ψ does not leak extra information to the adversary. This is because the entire Ψ is known to the adversary. For a binary search, the leaf determines the search path. Therefore, since the adversary observes the access to a white address in the database, it can generate the same trace as by T does.

The adversary also observes a retrieved black record and a search path from either **targeted-walk** or **random-walk**. Since the accessed black record's address can be inferred from the root-to-leaf walk in Γ, we only evaluate whether a tree walk leaks more query information than the touched black address. According to Lemma 4.1, the execution of **targeted-walk** in Algorithm 3 has the identical distribution for all queries. In addition to that, **random-walk** has the same path distribution as **targeted-walk** since every leaf node still has the same probability to be fetched. Therefore, neither tree walk algorithms leaks information about the query except the address of the black record to retrieve.

To summarize, the accesses to B, Ψ, Γ in Algorithm 3 do not leak more query information to the adversary than in Algorithm 1. □

Lemma 4.4: The accesses to auxiliary data structures $B, \hat{B}, L', \Gamma'$ in Algorithm 4 are oblivious.

PROOF. Algorithm 4 performs a full scan of B. Since B only contains black addresses which can be derived from database access pattern, the full scan of B does not leak extra information. The algorithm also reads $\hat{B}[v]$ in Line 11. This does not leak extra information, since the adversary can find $\hat{B}[v]$ by observing that $D_s[\hat{B}[v]]$ is fetched.

The construction of Γ' (including L') involves a comparison between the keys stored in two nodes. Although the adversary gets the topology of Γ' which reveals the order of all keys, the construction process does not leak more information than Γ' itself. In fact, Γ' leaks no more information to the adversary than she has observed. The y-values in L' are exactly the same as those in B, which can be easily obtained by observing the database retrievals. All other information in Γ' are the ciphertext of keys. Although Γ' shows the order of all keys, it does not expose the permutation used in the partial shuffle. This is because all keys in Γ''s inner nodes are randomly picked, and all $\overline{x}$-values in L' are derived from $H(x, v')$.

To summarize, any access pattern observed by the adversary can be caused by all possible permutations with the same probability. Thus, the accesses to $B, \hat{B}, L', \Gamma'$ do not leak more information to the adversary than in Algorithm 2. □

The lemmas have shown that accesses to auxiliary data structures leaks no additional information than in the basic construction. Since both the full scheme and the basic one have the same database access patterns for the query execution and partial shuffle, Lemma 3.1 and Lemma 3.2 still hold in the full scheme, which leads to the following theorem as in proving Theorem 3.1.

Theorem 4.1: For a database $D = [d_1, d_2, \cdots, d_n]$ and any corresponding shuffled and encrypted database D_s using pseudo-random permutations $\sigma, \pi_1, \cdots, \pi_s$ and a semantically secure encryption $\mathcal{E}$, for any PPT adversary $\mathcal{A}^{\mathcal{O}}$ against our scheme (Algorithm 3 and 4), there exists a PPT $\bar{\mathcal{A}}^{\mathcal{O}}$, such that for any clean query Q, $\forall K \in \mathbb{N}$, for any access pattern λ_K, $\forall q \in [1, n]$, $|\Pr(q \leftarrow \mathcal{A}_Q^{\mathcal{O}}(\lambda_K)) - \Pr(q \leftarrow \bar{\mathcal{A}}_Q^{\mathcal{O}}(1^\kappa))| < \epsilon(\kappa)$. The probability is taken over all the internal coin tosses of Algorithm 3, Algorithm 4, Q, and $\mathcal{A}$ or $\bar{\mathcal{A}}$.

V. PERFORMANCE

A. Complexity Analysis

Our scheme has an $\mathrm{O}(\log n)$ communication complexity, which is the same as that of schemes in [10], [9], [19], and is the lower bound of communication complexity for any PIR construction. The computational complexity describes the amount of accesses on H, including database accesses (e.g. read and write), auxiliary data structure accesses. Note that $k/2$ queries are executed in every session. When the i-th session starts, H holds $ik/2$ black records. Therefore, one query execution of Algorithm 3 costs $2(\log(ik/2) + 1) = 2(\log i + \log k)$ accesses due to the task of binary searches on Γ and Ψ. A partial shuffle at the end of the i-th session permutes $(i + 1)k/2$ black records. It requires $(i + 1)k/2$ accesses to scan array B, $4 \times (i + 1)k/2$ accesses to permute the records, $(i + 1)k/2 * \log((i + 1)k/2)$ accesses in average

for constructing L', $(i+1)k/2$ accesses for constructing Γ'. Therefore, totally $tk/2$ queries executed in t sessions costs $\sum_{i=1}^{t}[k(\log i + \log k) + \frac{1}{2}(i+1)k(\log(i+1) + \log k + 5)]$ server operations, which is approximated to:

$$C(t) \approx \frac{1}{4}kt^2 \log t + \frac{\log k + 15}{4}kt^2 + \frac{3}{2}kt \log t + \frac{4\log k + 15}{4}kt \quad (8)$$

Therefore, the complexity of the amortized server computation cost per query is $\mathrm{O}(t \log t)$, which is independent of the database size.

The advantage of our scheme decreases when t asymptotically approaching n. One optimization is to *reset* the state when t is large. A reset is to run a full shuffle on the original database which costs $2n$ acesses using the full shuffle algorithm in [19]. Let τ be the parameter such that the database is reset for every τ sessions. Then, the average amount of accesses $\overline{C}$ is

$$\overline{C} = \frac{C(\tau) + 2n}{k\tau/2} \approx \frac{1}{2}\tau \log \tau + \frac{\log k + 15}{2}\tau + 3\log \tau + \frac{4n}{k\tau}$$

We choose an optimal $\tau = \sqrt{\frac{16n}{k(\log n + \log k + 14)}}$, which satisfies that $\frac{1}{2}\tau \log \tau + \frac{\log k + 15}{2}\tau + 3\log \tau \approx 4n/k\tau$, such that the optimal average cost becomes

$$C^* = \sqrt{\frac{4n(\log n + \log k + 14)}{k}}$$

Thus, the complexity of the average computation cost per query after optimization is $\mathrm{O}(\sqrt{\frac{n(\log n + \log k)}{k}})$.

A comparison of our scheme against other PIR schemes is given in Table III. Note that all previous hardware-assisted schemes [9], [10], [18], [19] 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 $\mathrm{O}(n)$. Our scheme substantially outperforms all other PIR schemes in terms of average query cost by paying a slightly higher price of online query processes.

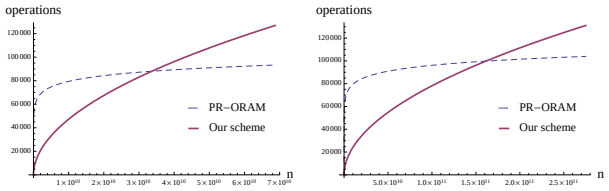
Schemes	**Runtime computation cost**	**Amortized computation cost**
Our scheme running t sessions	$\mathrm{O}(\log t + \log k)$	$\mathrm{O}(t \log t)$
Our scheme with reset for every $\sqrt{\frac{8n}{k(\log n - \log k + 2)}}$ sessions	$\mathrm{O}(\log t + \log k)$	$\mathrm{O}(\sqrt{\frac{n(\log n + \log k)}{k}})$
Scheme in [19]	$\mathrm{O}(1)$	$\mathrm{O}(n/k)$
Scheme in [9], [10]	$\mathrm{O}(1)$	$\mathrm{O}(\frac{n \log n}{k})$
Scheme in [18]	$\mathrm{O}(1)$	$\mathrm{O}(n)$
PIR schemes in the standard model	$\mathrm{O}(n)$	$\mathrm{O}(n)$

TABLE III
COMPARISON OF COMPUTATION COMPLEXITY IN TERMS OF THE AMOUNT OF SERVER ACCESSES.

B. Comparison with Hierarchy-based ORAM

We also compare our scheme with the state-of-the-art ORAM proposed in [15] (denoted by PR-ORAM). The comparison is made upon several aspects including computation complexity, the actual computation cost, the protected storage cost and the server storage cost.

- **Complexity.** Clearly, the $\mathrm{O}(\sqrt{n \log n/k})$ complexity of our scheme is much higher than the $\mathrm{O}(\log^2 n)$ complexity of PR-ORAM and other hierarchy-based ORAM constructions.
- **Actual Computation Cost.** According to [15], the constant factor in the big-O notation of PR-ORAM's server operation complexity is about 72 if two optimization techniques are applied. (Otherwise, it is about 160 according to their experiments.) Therefore, it takes $72\log^2 n$ operations per query. The average cost of our scheme with optimization is $\sqrt{\frac{4n(\log n + \log k + 14)}{k}}$. By conservatively setting $k = 1024$, our scheme outperforms PR-ORAM's $72\log^2 n$ operations when $n < 3 * 10^{10}$ as shown in Figure 3(a). Note that a popular trusted hardware, e.g. IBM PCIXCC, typically has megabytes storage, which can accommodate thousands of items. It is more suitable than PR-ORAM for databases of up to billions of items.



(a) k=1024 (b) k=4096

Fig. 3. Comparison between our scheme and ORAM[15] with different cache size

- **Protected Storage.** Both our scheme and PR-ORAM need a protected storage whose size is independent of the database size. In our scheme, the hardware needs a cache to store a constant amount of data items. PR-ORAM also needs a client end storage to store secret information. Since it does not store data, it requires less storage than our scheme.
- **Server-side Storage.** In our scheme, the server storage grows with query executions. In maximum, it stores the database of n items, two arrays B and $\hat{B}$ of size $2\sqrt{nk}$, a tree Γ (including L) of $4\sqrt{nk}$ nodes, and an array Ψ of $2\sqrt{nk}$. Therefore, the maximum storage cost at the server side is $n + 8\sqrt{nk}$ in maximum, in contrast to the $8n$ server storage in PR-ORAM.
- **Architecture.** Although we introduce a trusted hardware in the server side, the algorithms proposed in this paper can also be applied to client-server settings as ORAM-based PIR. We remark that to solve the PIR problem, both our scheme and ORAM require a trusted entity. In a tight-coupling architecture considered in our scheme, a secure hardware is the one, which supports multiple clients and has faster database accesses. In a loose-coupling

architecture as suggested in [21], a client/agent plays the role of trusted party. Note that the choice of architecture does not affect the complexity of the algorithms or the number of server operations.

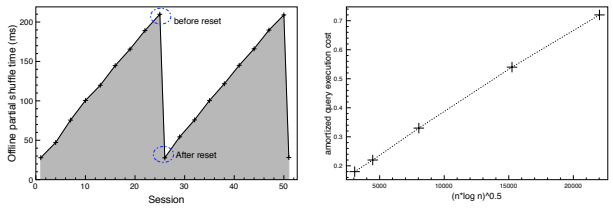
C. Experiment Results

We have implemented Algorithm 3 and 4 and measured their computation time cost with a simulated trusted hardware. Both algorithms are executed on a PC with a Pentium D CPU at 3.00GHz, 1GB memory, and Ubuntu 9.10 x86_64. They are implemented by using OpenSSL-0.9.8 library, where the permutation π_s is implemented using the generalized Feistel construction proposed by Black and Rogaway in [4].

Our experiment is to verify our square-root performance analysis in Section V-A. We fix the cache size as 512 items and experiment with databases of five different sizes. For each database, we ran 100,000 random generated queries with full-shuffles after a fixed amount of sessions. We measured the average query time in each experiment. The results are shown in Table IV below, and are plotted in Figure 4. Figure 4(a) depicts the up-and-down of the partial shuffle time, where the drop is due to the protocol reset. Figure 4(b) depicts the average query execution time growing almost *linearly* with $\sqrt{n \log n}$ which confirms our analysis above.

Database size (n)	500k	1m	3m	10m	20m
Online computation (μs)	39	39	40	40	40
Overall computation (ms)	0.18	0.22	0.33	0.54	0.72

TABLE IV
QUERY EXECUTION TIME FOR DIFFERENT DATABASE SIZES n, WITH A FIXED CACHE SIZE k=512.



(a) Effect of reset by full-shuffle (b) Linear relation between amortized query execution time and $\sqrt{n \log n}$

Fig. 4. Experiment results of the proposed PIR scheme

VI. CONCLUSION

We have presented a novel hardware-based scheme to prevent database access patterns from being exposed to a malicious server. By virtue of twin-retrieval and partial-shuffle, our scheme avoids full-database shuffle and reduces the amortized server computation complexity from $\mathrm{O}(n)$ to $\mathrm{O}(t \log t)$ where t is the number of queries, or to $\mathrm{O}(\sqrt{n \log n/k})$ with an optimization using reset. Although the hierarchy-based ORAM algorithm family [15], [20], [21] can protect access patterns with at most $\mathrm{O}(\log^2)$ cost, they are plagued with large constants hidden in the big-O notations. With a modest cache $k = 1024$, our construction outperforms those poly-logarithm algorithms for databases of $3 * 10^{10}$ entries. In addition, our scheme has much less server storage overhead. We have formally proved the scheme's security following the notion of private information retrieval and showed our experiment results which confirm our performance analysis.

ACKNOWLEDGEMENT

This research is partly supported by the Office of Research, Singapore Management University. We would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] T.W. Arnold and L.P. Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eserver. *IBM Journal of Research and Development*, 48, May 2004.
[2] A. Beimel, Y. Ishai, E. Kushilevitz, and Jean-François Raymond. Breaking the $o(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of IEEE FOCS'02*, pages 261–270, 2002.
[3] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of CRYPTO'00*, pages 55–73, 2000.
[4] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Proceedings of CT-RSA 2002*.
[5] B. Chor and N. Gilboa. Computationally private information retrieval. In *Proceedings of the 29th STOC'97*, pages 304–313, 1997.
[6] B. Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE FOCS'95*, pages 41–51, 1995.
[7] Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In *Proceedings of the 8th Australasian Conference on Information Security and Privacy*, 2003.
[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.
[9] A. Iliev and S Smith. Private information storage with logarithm-space secure hardware. In *Proceedings of International Information Security Workshops*, pages 199–214, 2004.
[10] A. Iliev and S. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security & Privacy*, 3(2):20–28, 2005.
[11] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally private information retrieval. In *Proceeding of the 38th IEEE FOCS'97*, pages 364–373, 1997.
[12] E. Kushilevitz and R. Ostrovsky. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In *Proceedings of Eurocrypt'00*, pages 104–121, 2000.
[13] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of ISC 2005*.
[14] R. Ostrovsky and V. Shoup. Private information storage. In *Proceedings of the 29th STOC'97*, pages 294 – 303, 1997.
[15] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Proceedings of CRYPTO 2010*.
[16] Victor Shoup. Sequence of games: A tool for taming complexity in security proofs. *Cryptology ePrint report 2004/332, November 30*, 2004.
[17] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of NDSS'07*, 2007.
[18] S. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001.
[19] S. Wang, X. Ding, R. Deng, and F. Bao. Private information retrieval using trusted hardware. In *Proceedings of the 11th ESORICS'06*, pages 49–64, 2006.
[20] P. Williams and R. Sion. Usable PIR. In *Proceedings of NDSS 2008*.
[21] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proceedings of ACM CCS 2008*.
[22] Y. Yang, X. Ding, R. Deng, and F. Bao. An efficient pir construction using trusted hardware. In *Proceedings of Information Security Conference*, 2008.

Xuhua Ding Xuhua Ding is an Assistant Professor in the School of Information Systems at Singapore Management University. He earned his Ph.D. degree in Computer Science from the University of Southern California and spent two years during his Ph.D. study at the University of California at Irvine. He received his B.Sc. and M.S. degrees in Computer Science and Engineering from Shanghai Jiao Tong University. His research areas include privacy-preserving algorithms, trustworthy computing, applied cryptography.

Yanjiang Yang Yanjiang Yang received his B.Eng. and M.Eng. in computer science and engineering from Nanjing University of Aeronautics and Astronautics, China, in 1995 and 1998, Ph.D. in information security and applied cryptography from National University of Singapore in 2005, respectively. He is now a research fellow in Institute for Infocomm Research, A*STAR, Singapore. His research areas include information security and applied cryptography.

Robert H. Deng Robert H. Deng received his Bachelor from National University of Defense Technology, China, his MSc and PhD from the Illinois Institute of Technology, USA. He has been with the Singapore Management University since 2004, and is currently Professor, Associate Dean for Faculty & Research, School of Information Systems. Prior to this, he was Principal Scientist and Manager of Infocomm Security Department, Institute for Infocomm Research, Singapore.

He has 26 patents and more than 200 technical publications in international conferences and journals in the areas of computer networks, network security and information security. He has served as general chair, program committee chair and program committee member of numerous international conferences. He is an Associate Editor of the IEEE Transactions on Information Forensics and Security, Associate Editor of Security and Communication Networks Journal (John Wiley), and member of Editorial Board of Journal of Computer Science and Technology (the Chinese Academy of Sciences).

He received the University Outstanding Researcher Award from the National University of Singapore in 1999 and the Lee Kuan Yew Fellow for Research Excellence from the Singapore Management University in 2006. He was named Community Service Star and Showcased Senior Information Security Professional by (ISC)2 under its Asia-Pacific Information Security Leadership Achievements program in 2010.