Adjusting the Trade-Off between Privacy Guarantees and Computational Cost in Secure Hardware PIR

Spiridon Bakiras[1] and Konstantinos F. Nikolopoulos[2]

[1] John Jay College, City University of New York
sbakiras@jjay.cuny.edu
[2] The Graduate Center, City University of New York
knikolopoulos@gc.cuny.edu

Abstract. Database queries present a potential privacy risk to users, as they may disclose sensitive information about the person issuing the query. Consequently, privacy preserving query processing has gained significant attention in the literature, and numerous techniques have been proposed that seek to hide the content of the queries from the database server. Secure hardware-assisted private information retrieval (PIR) is currently the only practical solution that can be leveraged to build algorithms that provide perfect privacy. Nevertheless, existing approaches feature *amortized* page retrieval costs and, for large databases, some queries may lead to excessive delays, essentially taking the database server offline for large periods of time. In this paper, we address this drawback and introduce a novel approach that sacrifices some degree of privacy in order to provide *fast* and *constant* query response times. Our method leverages the internal cache of the secure hardware to constantly reshuffle the database pages in order to create sufficient uncertainty regarding the exact location of an arbitrary page. We give a formal definition of the privacy level of our algorithm and illustrate how to enforce it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method can provide low page access times, even for very large databases.

1 Introduction

Internet users are becoming increasingly wary of the potential privacy risks associated with their everyday online activities. Web search engines, for example, maintain detailed logs of every query that they receive. However, with sophisticated data mining techniques, these query logs can reveal sensitive information about a user's lifestyle, health, habits, etc. [4,15]. Similarly, the emergence of location based services (LBS) allows mobile users to browse points of interest (e.g., restaurants) in their surroundings. Since these queries are also logged at the LBS provider, a user's location over a period of time can be tracked with very high accuracy [23].

Clearly, ordinary database queries involve an inherent privacy risk for users and, as a result, privacy preserving query processing is an emerging research

W. Jonker and M. Petković (Eds.): SDM 2011, LNCS 6933, pp. 128–144, 2011.

field in the database community. A popular approach that enhances the level of privacy in certain applications, is anonymity. The central principle of anonymity is to inject sufficient noise into a query, so that the user has *plausible deniability* over the exact content of the query. For instance, the client could combine the real query with several dummy ones (that are typically unrelated) or alter slightly the query parameters. Algorithms based on anonymity have been proposed for both text search engines (e.g., [21,22]) and location based services (e.g., [3,8,16,20]). However, since the database server has access to the plaintext queries, it may be able to determine the real content of a query using background knowledge (e.g., detailed information about a specific user).

Data encryption is another technique that can be leveraged to hide the content of a query. In this scenario, the server interacts with an encrypted version of the original database. Queries are also encrypted in a similar fashion and, thus, the server can not deduce any information about the query content. Research work is this area has focused on developing efficient encryption algorithms that facilitate exact query processing at the server side [1,2]. The limitation of encryption schemes, however, is that two identical queries always produce the same encrypted result. Consequently, if the server has knowledge of the access patterns of the database records (i.e., their relative popularities), it can extract some information about a query through the records included in the result set.

Private information retrieval (PIR) is the only solution available that can be leveraged to build algorithms that provide perfect privacy. In particular, PIR protocols [7] allow a client to retrieve any record from a database, while making it impossible for a computationally bounded server to determine which record was retrieved. Note that, when PIR is employed, the server cannot perform the actual query processing. Instead, the client accesses (privately) the disk-resident index structure at the database server and resolves the query locally through a series of PIR retrievals [23]. Currently, secure hardware-assisted PIR is the only *practical* PIR construction. It is implemented on top of a tamper-resistant CPU (secure hardware), which acts as a proxy between the server and the clients. Nevertheless, existing approaches feature *amortized* page retrieval costs, because they necessitate periodic reshuffle operations on the database. As a result, some queries may lead to excessive delays, essentially taking the database server offline for large periods of time.

In this paper, we address this drawback and introduce a novel approach that sacrifices some degree of privacy in order to provide *fast* and *constant* query response times. The goal is to design a system that balances the trade-off between computational cost and privacy guarantees. In other words, we aim to provide a much stronger notion of privacy compared to anonymity or encryption based schemes, but with a computational cost that is considerably lower compared to existing PIR techniques. Such a system would benefit applications that do not require perfect privacy, but are instead satisfied with a sufficient level of uncertainty.

Our algorithm initially encrypts and obliviously permutes the database pages. Each page is then retrieved efficiently by accessing (through the secure hardware)

its encrypted version from the server's disk. To further enhance the privacy of our approach, we introduce a randomized algorithm that constantly reshuffles the underlying pages in order to create sufficient uncertainty regarding the exact location of an arbitrary page. The algorithm works by randomly moving every requested page to a new location on the disk. In particular, it leverages a built-in cache at the secure hardware that stores a fixed number of previously retrieved pages. Reshuffling occurs during each page request, with a random page from the cache being written to a new location on the disk. We give a formal definition of the privacy level of this approach and illustrate how to enforce it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method can provide low page access times, even for very large databases. In summary, the contributions of our work are the following.

- We propose a novel architecture, based on state-of-the-art secure hardware, that reduces significantly the cost of private page retrievals compared to existing PIR based techniques.
- We formally define the privacy level of our approach and use analytical models to derive the corresponding security parameter.
- We evaluate the performance of our method, using (i) analytical results from a secure hardware deployment and (ii) measurements from a software implementation. We show that, given sufficient secure storage capacity, our system can achieve sub-second query response times, even for TB-sized databases.

The remainder of this paper is organized as follows. Section 2 reviews previous work on database privacy and private information retrieval techniques. Section 3 describes the architecture of our approach and outlines the underlying adversarial model. Section 4 introduces our private page retrieval algorithm and Section 5 presents the analytical results from a secure hardware implementation. Finally, Section 6 concludes the paper.

2 Related Work

PIR was first introduced by Chor et al. [7], and is formally defined as follows. The server holds a database, which is assumed to be a binary string X of length n. The client wants to retrieve the i-th bit (x_i) of the database, without the server knowing the value of the index i. In general, PIR protocols can be classified into three main categories: information theoretic, computational, and secure hardware.

First, *information theoretic* PIR [5,7,12,27] ensures that the query discloses no information about the retrieved bit, even if the server has unbounded computational power. However, these protocols are not practical, as they require that the database be replicated into k *non-colluding* servers. On the other hand, *computational* PIR protocols [6,10,18,19] work with a single server, and employ well known cryptographic primitives that guarantee query privacy for a computationally bounded server. Nevertheless, these protocols are extremely expensive

for large databases, as they require at least one modular multiplication for every bit of the database.

Finally, *secure hardware* PIR [14,24,25,26] relies on a tamper resistant CPU (located at the server side), which acts as a proxy between the clients and the server. These protocols are significantly faster than computational PIR, because they do not need to scan the whole database for every query. Wang et al. [24] utilize the internal storage of the secure hardware that can hold k out of n database pages. Every request inserts a new page into the secure storage and, when the storage capacity is reached, the database is reshuffled. Therefore, the amortized computational cost of this approach is $O(n/k)$. Ref. [14,25,26] leverage the Oblivious RAM model [13], which arranges the database pages into a pyramid-like structure. To achieve access pattern privacy, (i) every level of the structure is accessed during a page retrieval and (ii) the pyramid levels are periodically reshuffled by the secure hardware. Iliev and Smith [14] propose a method with $O(\sqrt{n}\log n)$ amortized computational cost, while Williams and Sion [25] improve this amortized cost to $O(\log^2 n)$. Currently, the state-of-the-art approach is due to Williams et al. [26], and provides an amortized logarithmic computational cost of $O(\log n \log\log n)$. However, due to the periodic reshuffling of the pyramid levels, the response time of a single PIR retrieval may vary from hundreds of milliseconds to thousands of seconds (as illustrated in [26]).

PIR based solutions have been explored previously in the context of spatial nearest neighbor queries. In particular, Khoshgozaran et al. [17] and Papadopoulos et al. [23] utilize secure hardware protocols, while Ghinita et al. [11] employ an expensive computational PIR algorithm [18]. Ref. [23] is a more general and comprehensive study on the applicability of PIR protocols on multi-level index structures. The authors introduce a solution that provides perfect privacy, and also present a detailed experimental evaluation based on secure hardware [25] simulations. Their results show that query processing may require tens of seconds, even for moderate databases, due to the large number of PIR retrievals on the underlying disk-resident index structures. Motivated by this fact, we propose an alternative approach that sacrifices some degree of privacy in order to reduce significantly the query processing cost.

3 Preliminaries

Section 3.1 describes the basic architecture of our approach and Section 3.2 outlines the underlying threat model.

3.1 System Architecture

Figure 1 illustrates the proposed system architecture. The secure hardware is a *tamper resistant* CPU, such as the IBM 4764 PCI-X secure coprocessor[1]. It is attached at the server machine, but it can be trusted to operate without

[1] `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`

any interference from the server. Specifically, it includes tamper detecting and responding circuitry that, in the event of an attack, destroys all the critical keys and certificates. The secure hardware incorporates an internal cache (up to 64MB for the IBM 4764 secure coprocessor) that is inaccessible by the server, and also has direct access to the server's disk.

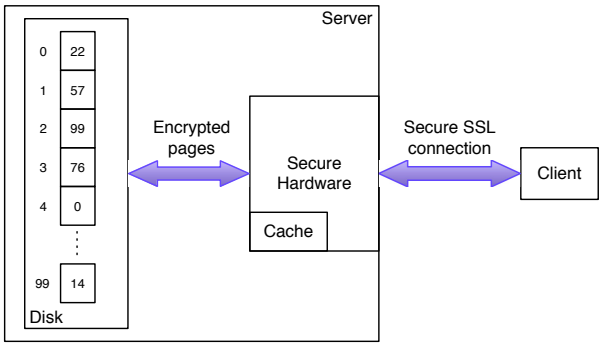


Fig. 1. System architecture

Note that the secure hardware is only necessary in the *three-party* querying model, i.e., when any client (including the adversary) is allowed to query the database server. Nevertheless, our methods are also applicable in the *two-party* querying model. The two-party model applies to the *database outsourcing* paradigm, where the data owner is the only client that accesses the database. In this setting, the owner outsources its data to a third-party service provider and wishes to access these data in a private manner. Since the data owner is the sole client in this architecture, there is no need for a secure hardware platform at the service provider. Instead, the functionality of the secure hardware can be implemented entirely at the owner's side (physically isolated from the adversary), using any standard server configuration. We explore the feasibility of this approach in Section 5.

In our problem formulation, we consider a database consisting of n pages (Table 1 summarizes the symbols used throughout the paper). Each page is a tuple $\langle id, data \rangle$, where the id attribute uniquely identifies the page. Prior to query processing, the secure hardware encrypts and obliviously permutes the database pages. It utilizes a *symmetric-key* encryption algorithm, such as AES [9], and the encryption key is secret from both the database server and the clients. Clients communicate with the secure hardware via secure SSL connections. A client query $Q(i)$ is simply a request to retrieve the page with $id = i$ from the database (we assume pages are assigned id values ranging from 0 to $n - 1$). To facilitate query processing, the secure hardware stores in its cache a look-up table that maps each page id to its actual position on the disk. After identifying the corresponding position, the secure hardware retrieves the page from the disk, decrypts it, and finally transmits it to the client via the secure connection.

Table 1. Summary of symbols

Symbol	Description
n	Database size (number of pages)
k	Block size (number of pages)
T	Number of blocks in database ($= n/k$)
m	Cache capacity (number of pages)
B	Page size (bytes)

To provide perfect query privacy, previous approaches apply periodically an oblivious permutation algorithm to reshuffle the database pages. Note that, after the reshuffling operation, every database page has an equal probability ($= 1/n$) of landing in any of the n available locations. Consequently, any query that accesses a new page from the disk becomes indistinguishable from any other query. In this work, we aim to relax this stringent constraint and allow pages to land in different disk locations according to a non-uniform distribution. Unlike prior methods, we do not reshuffle the entire database at once; instead, during each request instant, *one* previously retrieved page (that resides temporarily inside the cache) is relocated to a new position on the disk. In particular, for any value $c \geq 1$, we introduce the notion of *c-approximate PIR* as follows.

Definition 1. *A scheme provides* ***c-approximate PIR*** *if, after moving a single page p to a new location on the disk and for any pair of disk locations l_i, l_j, the probability of p landing in location l_i is at most c times larger than the probability of landing in location l_j.*

The value c is the privacy parameter of our approach, as it determines the variability of the distribution that models the individual page relocation process. Smaller values of c result in better privacy, while the case $c = 1$ offers perfect privacy (i.e., equivalent to PIR).

3.2 Adversarial Model

We assume that the adversary is the server itself, and its goal is to derive any non-trivial information regarding the *id* of a requested page. Because of the underlying secure SSL connections, both the client queries and the generated replies are unreadable by the server. Nevertheless, the server can see the accessed locations on the disk and has knowledge of all the algorithms that are implemented inside the secure hardware. We also assume that the server can only perform polynomial time computations and is "curious but not malicious" (i.e., it will not tamper will the actual data).

4 Private Page Retrieval Algorithm

Section 4.1 describes the page retrieval algorithm, while Section 4.2 provides an analytical model that quantifies its privacy level. Section 4.3 illustrates the database update procedure.

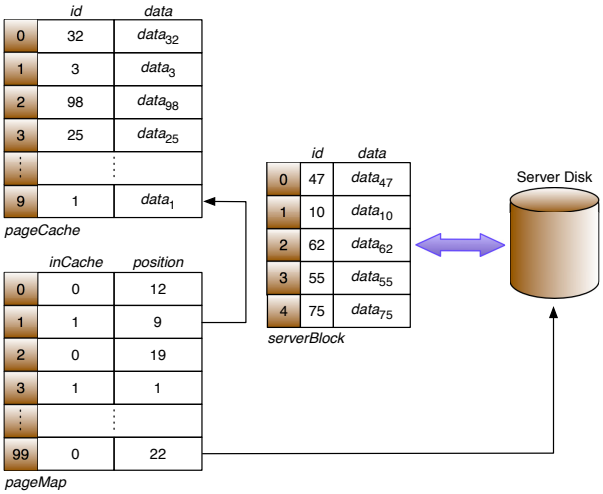


Fig. 2. Data structures at the secure hardware

4.1 Algorithm

Our approach leverages the built-in cache at the secure hardware to *obliviously mix* a pool of database pages and copy them into random positions on the disk. We assume that the cache can store a total of m pages and employs a *randomized* cache replacement policy. Note that the purpose of the cache is not to improve the page retrieval time, but to facilitate this *continuous* page reshuffling process.

During each page request, the algorithm retrieves a fixed number of $k+1$ pages, where k is the security parameter. In particular, the secure hardware initially reads (in a round-robin manner) a *block* of k contiguous pages. Specifically, on the first request it accesses the database pages at locations 0 through $k-1$, next the pages at locations k through $2k-1$, etc. The $(k+1)$-th page that is read is either the page requested by the client or a random one (the detailed algorithm is explained shortly). The reason for reading multiple pages is to guarantee that any cached page has a non-negligible probability of being written to any location on the disk (discussed in Section 4.2). If n is not a multiple of k, the secure hardware inserts an appropriate number of dummy pages during the initial reshuffling stage.

Figure 2 shows the data structures maintained at the secure hardware. First, the cache is implemented as a vector (*pageCache*) holding m database pages. *pageMap* is a vector of size n and corresponds to the look-up table for all the database pages. Each entry in *pageMap* is a tuple $\langle inCache, position \rangle$. Attribute *inCache* uses a single bit that, when set, indicates that the corresponding page is stored inside the cache. Attribute *position* is an integer value that has a dual interpretation: if $inCache = 1$, it represents the index in the *pageCache* vector where the page is stored; otherwise, it identifies the location of that page at the server disk under the current permutation order (see Figure 2). Finally, *serverBlock* is the vector (of size $k+1$) that temporarily stores the pages that

Retrieve(i)

```
    // read next block (of size k) in a round-robin fashion
1:  serverBlock[0..k − 1] ← read(nextBlock)
2:  if (pageMap[i].inCache or i ∈ serverBlock)
      // select a random page that is not cached
      // and is not retrieved in serverBlock
3:    do
4:      p ← random(0, n − 1)
5:    while (pageMap[p].inCache or p ∈ serverBlock)
6:    if (pageMap[i].inCache)
7:      result ← pageCache[pageMap[i].position]
    // else use requested page
8:  else
9:    p ← i
    // read page p from the disk
10: serverBlock[k] ← read(pageMap[p].position)
    // decrypt all pages in serverBlock
11: decrypt(serverBlock)
12: if (!pageMap[i].inCache)
13:   q ← index of page i in serverBlock
14:   result ← serverBlock[q]
15: else
16:   q ← k
    // select a random page from the block
17:   r ← random(0, k − 1)
18: swap(serverBlock[r], serverBlock[q])
    // select random page from cache
19: s ← random(0, m − 1)
20: swap(pageCache[s], serverBlock[r])
    // re-encrypt all pages in serverBlock (with a new nonce)
21: encrypt(serverBlock)
    // write updated pages at the disk
22: write(serverBlock)
    // update pageMap (3 pages)
23: update(pageMap[pageCache[s]])
24: update(pageMap[serverBlock[r]])
25: update(pageMap[serverBlock[q]])
    // send page i to the client over the SSL connection
26: return result
```

Fig. 3. The page retrieval algorithm

are written to or read from the disk. In the sample configuration of Figure 2, $n = 100$, $m = 10$, and $k = 4$.

The page retrieval algorithm is shown in Figure 3, and operates as follows. First, the client sends a query to the secure hardware, containing the *id* of the required page (e.g., page i). The secure hardware then reads and stores into *serverBlock* the next block of k pages, according to the round-robin schedule. Next, it accesses $pageMap[i]$ and identifies the current location of that page. If page i is located at the server and is not included in the *serverBlock* vector, the page is retrieved from the corresponding location on the disk and stored into *serverBlock*. If, on the other hand, page i is included in the *serverBlock* vector, the secure hardware selects a random page from the database that is not currently cached or stored into *serverBlock*.

Subsequently, the secure hardware decrypts all $k+1$ pages in *serverBlock* and extracts the requested page. It then selects a random page from the cache and replaces it with the newly requested page i. Similarly, the cached page is copied

into *serverBlock*, overwriting page i. However, to ensure that the cached page is moved to any of the k locations in the read block (corresponding to the first k pages in *serverBlock*) with equal probability, the requested page initially swaps places with a random page in the block (line 18). Next, the pages in *serverBlock* are re-encrypted with a *new* random nonce, and are eventually transferred back to the server's disk. Finally, the secure hardware modifies the necessary entries (for the swapped pages) at the $pageMap$ vector.

In the case where the requested page produces a cache hit, the secure hardware retrieves a random page p from the disk and repeats the same steps as above, i.e., as if page p was requested by the client. To summarize, during every query, the requested page (or a random page, in the case of a cache hit) is stored into the cache and a random page from the cache is moved to one of the k locations in the block that was accessed as part of that request. Note that, due to the randomized cache replacement policy, a certain cached page may be evicted while it is being requested by the client. Also, to avoid timing attacks, a cached page is not returned immediately to the client, because that would reveal the cache hit to the adversary.

4.2 Security Analysis

The page retrieval algorithm works by spreading the accesses for a single page over multiple disk locations. Once a page is requested and moves into the cache, it will be relocated to a new position during a subsequent request. Consequently, an adversary can only track probabilistically the location of an arbitrary page within the server's disk. Our goal is to properly adjust the block size k, in order to meet the privacy requirements of the c-approximate PIR definition (Section 3.1).

Consider a sequence of client requests at instants $t = 0, 1, 2, \ldots$. Assume that page p is copied into the cache during a client request at $t = 0$. Then, the probability that it moves back to the disk at time $t \geq 1$ is computed as:

$$P^t = \left(1 - \frac{1}{m}\right)^{t-1} \cdot \frac{1}{m} \tag{1}$$

Therefore, if the secure hardware accesses a set of k locations (from a single block) $\mathcal{L}_t = \{l_1, l_2, \ldots, l_k\}$ during the request at time t, the probability that page p is relocated to position l_j $(1 \leq j \leq k)$ is equal to:

$$P^t_{p \rightarrow l_j} = \left(1 - \frac{1}{m}\right)^{t-1} \cdot \frac{1}{m} \cdot \frac{1}{k} \tag{2}$$

The value k is the security parameter of our approach, since it controls the time interval $T = n/k$ (given as number of requests) that is required to scan every location on the disk exactly once through the round-robin schedule. Note that Equation (2) is a monotonically decreasing function, so the k locations that are accessed at $t = 1$ have the highest probability of hosting page p. Specifically, for

the locations $l_j \in \mathcal{L}_1$, the probability that p is relocated there is:

$$P^1_{p \to l_j} = \sum_{i=0}^{\infty} \left(1 - \frac{1}{m}\right)^{T \cdot i} \cdot \frac{1}{m} \cdot \frac{1}{k} \tag{3}$$

Similarly, the locations $l_j \in \mathcal{L}_T$ have the lowest probability of storing page p:

$$P^T_{p \to l_j} = \sum_{i=0}^{\infty} \left(1 - \frac{1}{m}\right)^{(i+1) \cdot T - 1} \cdot \frac{1}{m} \cdot \frac{1}{k} \tag{4}$$

Consequently, the value of k can be determined by setting

$$\frac{P^1_{p \to l_j}}{P^T_{p \to l_j}} = \frac{1}{\left(1 - \frac{1}{m}\right)^{T-1}} = \frac{1}{\left(1 - \frac{1}{m}\right)^{\frac{n}{k}-1}} = c \tag{5}$$

Solving the above equation, we get:

$$k = \frac{n}{\frac{\log(1/c)}{\log(1-1/m)} + 1} \tag{6}$$

Note that, the value $c = 1$ corresponds to the trivial case of PIR, i.e., when the whole database is read for every request ($k = n$). On the other hand, a value such as $c = 2$ would indicate that any location is *at most* twice as likely to host a previously cached page as any other location on the disk. For a given database size n and privacy parameter c, the value of the security parameter k is determined by the available cache capacity. As evident in Equation (5), for a fixed value of T, the privacy parameter c converges towards 1 as the value of m increases.

4.3 Database Updates

A final remark concerns the handling of database updates in our system architecture. Similar to query processing, the database owner interacts only with the secure hardware through a secure SSL connection. Our system can handle trivially any type of updates, including insertions, deletions, and page modifications. In particular, every database update is treated as a regular query, i.e., the secure hardware (i) retrieves $k + 1$ pages from the disk, (ii) swaps one page from the cache with one of the retrieved pages, and (iii) writes the $k + 1$ pages back to the disk after re-encrypting them. Consequently, the type of update operation performed on the database is kept secret from the server.

Deletions are handled as cache hits, i.e., the $(k + 1)$-th page is selected randomly. Additionally, if the deleted page is stored inside the cache, it is always selected to swap positions with one of the k pages in the block. Finally, the *position* attribute of the $pageMap$ entry for that page is set to a reserved value (e.g., all 1's) that signifies the deletion event. Note that, if there are numerous page deletions on the database, the owner may choose to reshuffle (offline) the

whole database in order to physically remove the deleted pages. Page modifications are handled as regular queries, i.e., they can either produce a cache hit (if the page is stored inside the cache) or a cache miss. In any case, the original page is replaced with the new version.

To handle insertion operations, the secure hardware should reserve in advance sufficient storage space in its internal data structures. Therefore, during the initial reshuffling stage, the secure hardware should create numerous dummy pages that may be utilized to store the newly inserted pages. These pages are marked as *deleted*, so pages that are explicitly deleted by the data owner may serve the same purpose as well. When a new page is created in the database, the secure hardware accesses the next block of k pages as usual. However, the $(k+1)$-th page is always a *deleted* page. The newly inserted page is then stored inside the cache, replacing one of the pages therein. Finally, the deleted page swaps positions with one of the k locations of the retrieved block, and the evicted page is copied over the deleted page.

5 Secure Hardware Deployment

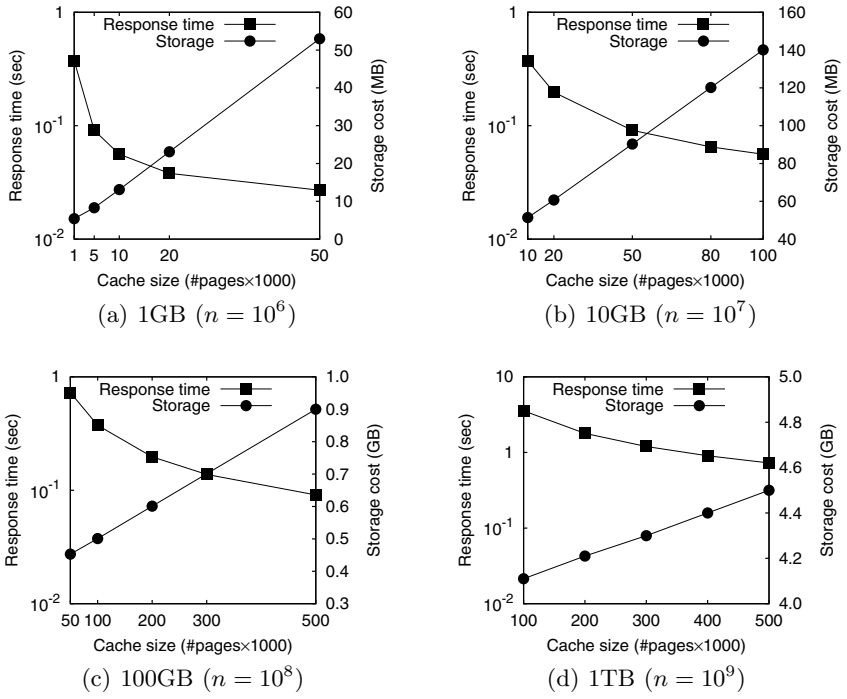
In this section we analyze the storage requirements and query processing cost of our methods in a secure hardware deployment. Our analysis is based on the configuration shown in Table 2, which is similar to the ones assumed in related studies [23,25].

Table 2. System specifications

Parameter	**Value**
Secure hardware cache	64MB
Disk seek time (t_s)	$5ms$
Disk read/write (r_d)	100 MB/s
Secure hardware link bandwidth (r_b)	80 MB/s
Encryption/decryption (r_{ed})	10 MB/s

Secure Storage Requirements. The page retrieval algorithm necessitates the storage of the three vectors depicted in Figure 2, inside the secure hardware cache. Given a database of n pages, each of size B bytes, the *pageCache* vector stores exactly m pages, thus consuming $m \cdot B$ bytes. The *serverBlock* vector stores the $k+1$ pages that are read from the server, i.e., it requires $(k+1) \cdot B$ bytes of space. Finally, the *pageMap* vector maintains information about the position of all database pages, plus an additional bit that indicates whether a page is currently cached. Consequently, it requires $n \cdot (\log n + 1)$ bits of storage space. Summarizing, the total storage cost of our approach (in bytes) is given as:

$$S = n \cdot \left\lceil \frac{(\log n + 1)}{8} \right\rceil + (m + k + 1) \cdot B \tag{7}$$



(a) 1GB ($n = 10^6$) (b) 10GB ($n = 10^7$)

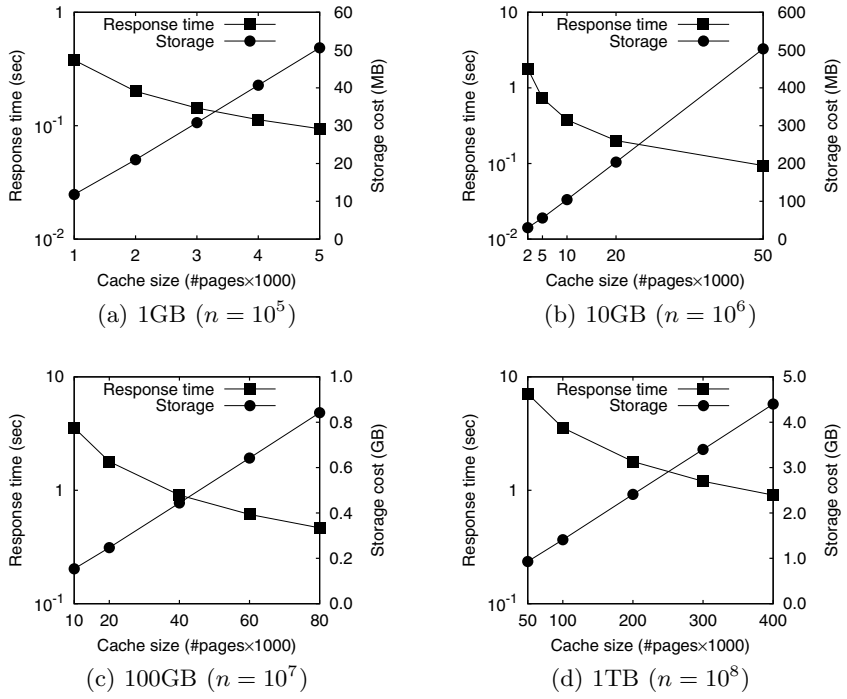
(c) 100GB ($n = 10^8$) (d) 1TB ($n = 10^9$)

Fig. 4. Page retrieval costs for 1KB pages ($c = 2$)

Page Retrieval Cost. For every client query the secure hardware needs to perform 4 random accesses at the server's disk. Two of those correspond to the read operations (one for reading the next block, and one for the additional page), while the remaining two are performed for writing back the re-encrypted pages. The $k + 1$ accessed pages are transferred twice between the secure hardware and the server (read/write) and are also processed twice by the encryption/decryption circuitry inside the secure hardware. Therefore, the query processing time at the server for retrieving a single page from the disk is:

$$Q_t = 4 \cdot t_s + 2 \cdot (k + 1) \cdot B \cdot \left(\frac{1}{r_d} + \frac{1}{r_b} + \frac{1}{r_{ed}} \right) \tag{8}$$

Figures 4 and 5 show some sample configurations for retrieving 1KB and 10KB pages, respectively, from databases of different sizes (with a privacy parameter $c = 2$). Specifically, they depict the page retrieval times and storage space requirements at the secure hardware as a function of the cache size m. For a 1GB database, a single secure coprocessor can retrieve privately 1KB pages in $27ms$ and 10KB pages in $94ms$. Note that, unlike existing secure hardware PIR schemes that feature *amortized* cost, the processing times shown here are *constant*. For larger databases, we may leverage multiple coprocessors at the server site to increase the secure storage capacity. This will boost the value of m, thus



(a) 1GB ($n = 10^5$) (b) 10GB ($n = 10^6$)

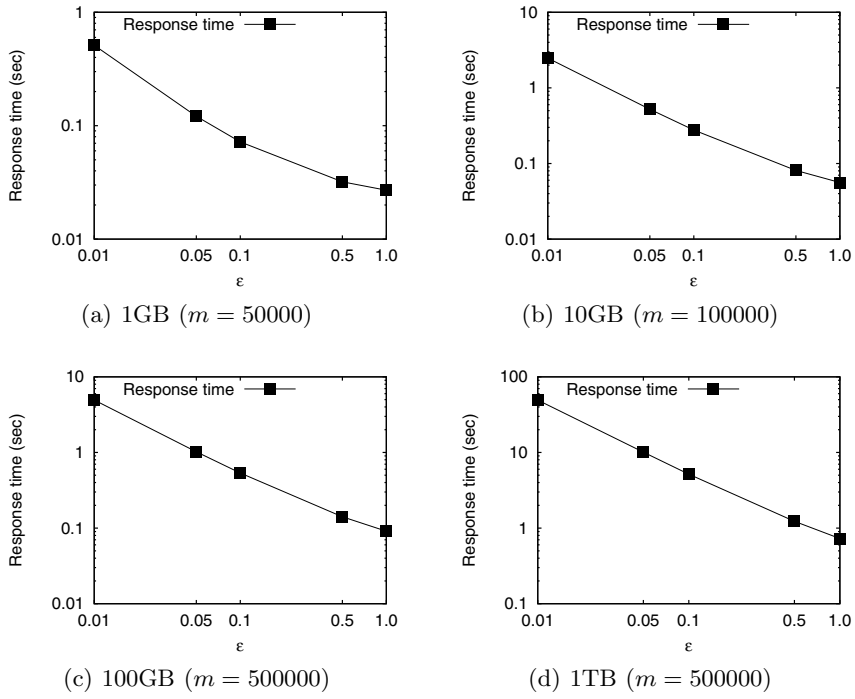
(c) 100GB ($n = 10^7$) (d) 1TB ($n = 10^8$)

Fig. 5. Page retrieval costs for 10KB pages ($c = 2$)

reducing considerably the security parameter k. For instance, with 1 coprocessor (up to 64MB of storage space) and a 10GB database, we can retrieve 1KB pages in 197ms and 10KB pages in 731ms. On the other hand, combining the storage space of 2 coprocessors can reduce those times to 65ms and 378ms, respectively.

Larger databases cannot be trivially handled by the current technology of tamper-resistant CPUs, due to the minimal storage resources that they provide. Consequently, 100GB databases will require 10 coprocessors to retrieve 1KB pages in 197ms and 10KB pages in 613ms. Even though this is an entirely feasible solution, it may increase considerably the monetary cost of PIR. Finally, for 1TB databases, sub-second page retrieval times (727ms for 1KB pages and 907ms for 10KB pages) are only feasible with over 4GB of secure storage. With the current technology, this capacity translates to over 70 coprocessor units. This excessive cost is mainly due to the *pageMap* data structure that maintains the location of every database page on the disk. However, this is an unavoidable cost because, unlike previous approaches that use hash functions to permute the entire database, our scheme reshuffles on a per page level and necessitates each page to be stored individually.

Figure 6 depicts the query response time as a function of the privacy parameter $c = 1 + \varepsilon$. We consider 1KB pages and set the cache sizes for the different databases to their largest values shown in Figure 4. Clearly, there is a



(a) 1GB ($m = 50000$) (b) 10GB ($m = 100000$)

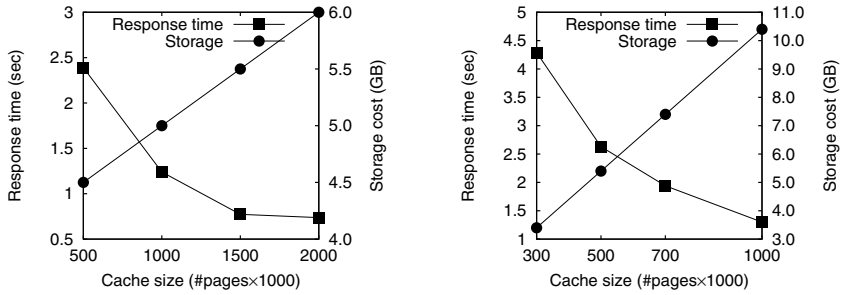
(c) 100GB ($m = 500000$) (d) 1TB ($m = 500000$)

Fig. 6. Response time as a function of $c = 1 + \varepsilon$ ($B = 1$KB)

trade-off between the privacy level of our approach and the computational cost. If we wish to provide better privacy, we need to retrieve more pages per block (increase k) in order to reduce the value of T. As shown in Equation (5), this will essentially decrease the value of the privacy parameter c. Nevertheless, our algorithm is efficient under strict privacy requirements and, for databases up to 100GB, sub-second query response times are achievable even for $c = 1.1$.

Despite the restrictions of current secure hardware technology, our methods are also applicable in the two-party querying model, as explained in Section 3.1. In this setting, the functionality of the secure hardware can be implemented on a powerful server (physically isolated from the adversary), thus allowing for much larger cache sizes. Consequently, our page retrieval algorithm can be implemented efficiently even for TB-sized databases. To verify the efficiency of this approach, we measured the page retrieval costs from a real implementation[2] of the two-party model. We set up the service provider and the owner to run on two different machines that were connected through a WiFi network. The network round-trip time (RTT) was set to $50ms$ and was simulated with the `sleep` function. Figure 7 illustrates the query response time and storage cost at the data

[2] We used the `Boost.Asio` library for the networking primitives and the `Crypto++` library for the AES implementation.



(a) 1KB pages ($n = 10^9$) (b) 10KB pages ($n = 10^8$)

Fig. 7. Page retrieval costs for 1TB database ($c = 2$)

owner as a function of the cache size m. With 6GB of storage space, the system can accommodate 2 million pages in its cache, achieving a query response time of $0.737ms$ (for 1KB pages). Note that the bottleneck in this architecture is the network transfer cost, since our algorithm necessitates the transfer of $(k + 1)$ database pages *twice* between the owner and the service provider. As a result, retrieving larger pages (10KB) requires a significant amount of storage space (to reduce the value of the security parameter k) and, as shown in Figure 7(b), over 10GB of space is necessary to achieve a query response time of $1.3s$.

6 Conclusions

Privacy preserving query processing is an emerging research field in the database community, due to the increasing demand for protecting user privacy. Existing techniques fail to provide adequate solutions, because they do not achieve a good trade-off between computational cost and privacy guarantees. On one hand, anonymity and encryption based schemes are computationally efficient, but they provide weak privacy. On the other hand, private information retrieval techniques offer perfect privacy, but their high computational cost renders them impractical for large databases. In this paper, we introduce a novel approach that provides a much stronger notion of privacy compared to anonymity or encryption based schemes, but with a computational cost that is considerably lower compared to existing PIR approaches. Our methods are built on top of a secure hardware that acts as a proxy between the clients and the server. The secure hardware encrypts and constantly reshuffles the database pages, in order to create sufficient uncertainty regarding the exact location of an arbitrary page. We give a formal definition of the privacy level of our algorithm and illustrate how to apply it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method is computationally efficient, even for very large databases.

Acknowledgments. This research has been funded by the NSF CAREER Award IIS-0845262.

References

1. Agrawal, D., Abbadi, A.E., Emekçi, F., Metwally, A.: Database management as a service: Challenges and opportunities. In: ICDE (2009)
2. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: SIGMOD (2004)
3. Ardagna, C.A., Cremonini, M., Damiani, E., di Vimercati, S.D.C., Samarati, P.: Location privacy protection through obfuscation-based techniques. In: DBSec (2007)
4. Barbaro, M., Zeller, T.: A face is exposed for AOL searcher no. 4417749. The New York Times (August 9, 2006)
5. Beimel, A., Ishai, Y., Kushilevitz, E., Raymond, J.E.: Breaking the $O(n^{1/(2k?-?1)})$ barrier for information-theoretic private information retrieval. In: FOCS (2002)
6. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, p. 402. Springer, Heidelberg (1999)
7. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: FOCS (1995)
8. Duckham, M., Kulik, L.: Simulation of obfuscation and negotiation for location privacy. In: Cohn, A.G., Mark, D.M. (eds.) COSIT 2005. LNCS, vol. 3693, pp. 31–48. Springer, Heidelberg (2005)
9. Garrett, P.: Making, Breaking Codes: Introduction to Cryptology, 1st edn. Prentice-Hall, Englewood Cliffs (2001)
10. Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 803–815. Springer, Heidelberg (2005)
11. Ghinita, G., Kalnis, P., Khoshgozaran, A., Shahabi, C., Tan, K.L.: Private queries in location based services: Anonymizers are not necessary. In: SIGMOD (2008)
12. Goldberg, I.: Improving the robustness of private information retrieval. In: IEEE Symposium on Security and Privacy (2007)
13. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)
14. Iliev, A., Smith, S.: Private information storage with logarithmic-space secure hardware. In: i-NetSec (2004)
15. Jones, R., Kumar, R., Pang, B., Tomkins, A.: I know what you did last summer: Query logs and user privacy. In: CIKM (2007)
16. Kalnis, P., Ghinita, G., Mouratidis, K., Papadias, D.: Preventing location-based identity inference in anonymous spatial queries. TKDE 19(12), 1719–1733 (2007)
17. Khoshgozaran, A., Shahabi, C., Shirani-Mehr, H.: Location privacy: Going beyond k-anonymity, cloaking and anonymizers. In: KAIS (2010)
18. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: FOCS (1997)
19. Lipmaa, H.: An oblivious transfer protocol with log-squared communication. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 314–328. Springer, Heidelberg (2005)
20. Mokbel, M.F., Chow, C.Y., Aref, W.G.: The New Casper: Query processing for location services without compromising privacy. In: VLDB (2006)
21. Murugesan, M., Clifton, C.: Providing privacy through plausibly deniable search. In: SDM (2009)

22. Pang, H., Ding, X., Xiao, X.: Embellishing text search queries to protect user privacy. PVLDB 3(1), 598–607 (2010)
23. Papadopoulos, S., Bakiras, S., Papadias, D.: Nearest neighbor search with strong location privacy. PVLDB 3(1), 619–629 (2010)
24. Wang, S., Ding, X., Deng, R.H., Bao, F.: Private information retrieval using trusted hardware. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 49–64. Springer, Heidelberg (2006)
25. Williams, P., Sion, R.: Usable PIR. In: NDSS (2008)
26. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: CCS (2008)
27. Woodruff, D.P., Yekhanin, S.: A geometric approach to information-theoretic private information retrieval. In: IEEE Conference on Computational Complexity (2005)