PRIVATE INFORMATION STORAGE WITH LOGARITHMIC-SPACE SECURE HARDWARE

Alexander Iliev and Sean Smith
Department of Computer Science, Dartmouth College
{sasho,sws}@cs.dartmouth.edu

Abstract In Private Information Retrieval (PIR), a user obtains one of N records from a server, without the server learning what record was requested.

Recent research in "practical PIR" has limited the players to the user and server and limited the user's work to negotiating a session key (eg. as in SSL)—but then added a secure coprocessor to the server and required the secure coprocessor to encrypt/permute the dataset (and often gone ahead and built real systems).

Practical PIR (PPIR) thus consists of trying to solve a privacy problem for a large dataset using the small internal space of the coprocessor. This task is very similar to the one undertaken by the older Oblivious RAMs work, and indeed the latest PPIR work uses techniques developed for Oblivious RAMs. Previous PPIR work had two limitations: the internal space required was still $O(N \lg N)$ bits, and records could only be read privately, not written.

In this paper, we present a design and experimental results that overcome these limitations. We reduce the internal memory to $O(\lg N)$ by basing the pseudorandom permutation on a Luby-Rackoff style block cipher, and by redesigning the oblivious shuffle to reduce space requirements and avoid unnecessary work. This redesign yields both a time and a space savings. These changes expand the system's applicability to larger datasets and domains such as private file storage.

These results have been implemented for the IBM 4758 secure coprocessor platform, and are available for download.

Keywords: Private information retrieval and storage, oblivious RAM, permutation network, sorting network, Luby-Rackoff cipher

1. INTRODUCTION

Private Information Retrieval (PIR) is a privacy-enhancing technique which has been receiving considerable research exploration, both theoretical and practical. The technique allows a user to retrieve data from a server without the server being able to tell what data the user obtained. It is of interest as a counterbalance to the increasing ease of collecting and storing information about a person's online activities, especially as these activities become a significant part of the person's life.

Examples of where PIR can be useful abound, usually where traffic analysis of encrypted data can yield useful information. A medical doctor retrieving *medical records* (even if encrypted) from a database may reveal that the owner of the record has a disease in which the doctor specializes. A company retrieving a patent from *a patent database* may reveal that they are pursuing a similar idea. Clients of both databases would benefit from the ability to retrieve their data without the database being able to know what they are interested in.

Two rather separate tracks exist in the PIR research record—one focuses on designing cryptographic protocols which achieve PIR by either making use of having the dataset on multiple non-communicating servers [3], or by using techniques based on intractability assumptions without multiple servers [2, 9].

The other track attempts to produce *Practical* PIR schemes [1, 7, 18] that can be integrated into existing infrastructure, by limiting the scheme to the server, and only requiring the client to negotiate a secure session to the server, as is typical in SSL sessions. This is made possible by using a physically protected space at the server—a *Secure Coprocessor* (SCOP) [17].

1.1 Existing Prototype

Our previous work on Practical PIR (PPIR) [7] produced a PPIR prototype running on the IBM 4758 secure coprocessor with Linux [17], and offering an LDAP[1] interface to the outside. We will first describe the background items related to this prototype.

Secure Coprocessors. A secure coprocessor is a small general purpose computer armored to be secure against physical attack, such that code running on it has some assurance of running unmolested and unobserved [22]. It also includes mechanisms to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [16]. The coprocessor is attached to a *host* computer. The SCOP is assumed to be trusted by clients (by virtue of all the above provisions), but the host is not trusted (not even its root user). The strongest adversary against the schemes presented here is the superuser on the host.

IBM 4758 Secure Coprocessor. The 4758 is a commercially available device, validated to the highest level of software and physical security

[1]Lightweight Directory Access Protocol—the protocol of choice for interfacing to online directories.

scrutiny currently offered—FIPS 140-1 level 4 [19]. It has an Intel 486 processor at 99 MHz, 4MB of RAM and 4MB of FLASH memory. It also has cryptographic acceleration hardware. It connects to its host via PCI (hence we often refer to it as a *card*). Our host runs Debian Linux, with kernel version 2.4.2-2 from Redhat 7.1 as needed by the 4758/Linux device driver.

In production, the 4758 runs the CP/Q++ embedded OS; however, experimental research devices can run a version of Linux (as does the follow-on product from IBM). Linux has considerable advantages in terms of code portability and ease of development—our prototype is written in C++, making extensive use of its language features and the Standard Template Library, and it runs fine on the 4758 with Linux.

PIR using Secure Coprocessors. The model which we follow is that we have available a physically protected computing space at the server. If this space was large enough to hold the whole dataset, the problem would be solved, as clients could negotiate a secure session with it, and then retrieve their data. Since it is physically protected, no one should be able to observe what item the client obtained. Unfortunately practical considerations result in real protected environments being quite small, much too small to hold the entire dataset. Thus, the problem becomes that we want to provide private access to a large dataset while using only a small amount of protected space. This is almost isomorphic to the Oblivious RAM problem [6], which we discuss further in Section 2.

Model. In Figure 1 we show the more concrete setup: we have a dataset of N named items each of size M. The items may be visible to the host; they may also be encrypted (for the SCOP's private key), though why and how they may be encrypted ahead of time is orthogonal to our topic here. A client connects to the SCOP (tunneling via the host) and delivers a request for one of the items. The SCOP is very limited in memory—it is allowed $O(\lg N + M)$ memory, which is the minimum needed to store pointers into the dataset, as well as a constant number of actual data items. Any larger storage, like the actual dataset or pre-processed versions of it, is provided by the host. Thus the SCOP has to make I/O requests to the host in order to service a client request. To be a correct PIR scheme, it must be the case that the host cannot learn anything[2] about client requests from observing the I/O from the SCOP.

Simply encrypting the records does not solve the problem; the server can still learn the *identity* of requested items, and (if the server colludes with a user) can learn what any given record decrypts to. It is also insufficient to only hide

[2]We are assuming that cryptography works; strictly speaking, this scheme is not secure in the information-theoretic sense, since the host can still see ciphertext.

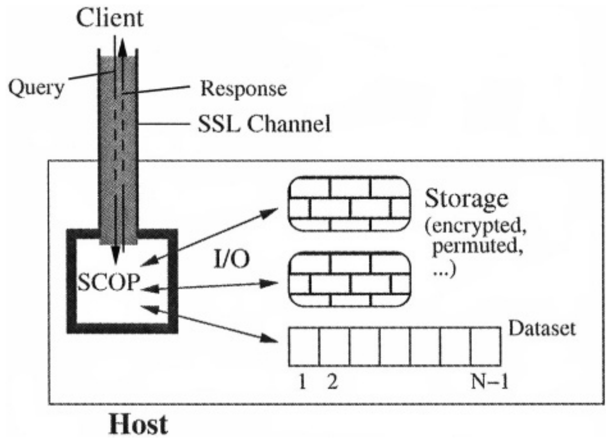

Figure 1. The setup of hardware assisted PIR

the identity of *single* retrievals, as then an attacker could learn the popularity of individual items, and correspondence between requests, eg. “Aphrodite and Boris both retrieved the same data item today”.

The Initial PIR with secure coprocessors algorithm. In their initial proposal of using secure hardware for PIR, Smith and Safford kept the dataset unprocessed on the host [18]. Given a request for item i, the SCOP reads *every* item in the dataset, internally keeps item i and returns it to the client at the end. The host only observes that the SCOP touched every record, so it does not learn anything about i. The clear problem is that every retrieval takes $O(N)$ time. (Careful data structures can permit the work to be divided evenly across several devices, but this time bound is still problematic.)

Latest PIR Algorithm. The structure of the algorithm we use was originally developed by Goldreich and Ostrovsky for the Oblivious RAM problem [6]. We note first that it relies on having a dataset of *numbered* items, from 1 to N. It proceeds in retrieval *sessions,* where a session S consists of:

Randomly permuting the contents of records 1 **through** N. First, the SCOP encrypts each record in the dataset. Then, the SCOP (pseudo)randomly selects a permutation π of $[1..N]$, and relocates the contents of each record r, $1 \leq r \leq N$, to record location $\pi(r)$, changing the encryption along the way. This produces the shuffled[3] dataset of encrypted items D_π. The relocations must

[3]We use permute and shuffle interchangeably, but shuffle always refers to permuting the whole dataset, as opposed to computing $\pi(i)$ for some i

be done so that the host cannot learn which permuted record corresponds to which input record, after having observed the pattern of record accesses during the permutation. Using the terminology of Goldreich et al., the permutation algorithm must be *oblivious:* have the same I/O access pattern regardless of the input (ie. the permutation)[4]. [6].

Servicing $k \ll N$ retrievals. By now, the permuted dataset D_π is available on the host, and the SCOP knows π. The SCOP uses this knowledge to hide the identities of retrieved records. In order to retrieve record *r,* the SCOP reads in $\pi(r)$ from D_π, and the host does not learn what *r* can be.

What is left is to hide the relationships between retrieved items, so the host (for example) cannot tell how many times a given item was retrieved. The approach is to copy records which have been accessed into a *working pool* P_S of maximum size *k,* which is scanned in its entirety for every retrieval. On each retrieval for record *r,* one record from D_π is added to P_S: either *r* if it is not already there, or a random untouched record if it is. Thus, records in D_π are accessed at most once.

The implementer can set a a maximum value of *k,* to put a maximum value on the response time for any given query. However, the shuffling step needs to be fast enough to have a new shuffle ready when P_S reaches that maximum *k.*

The private shuffle implementation has varied in the literature, and in our prototype we had added a new approach: using Beneš *permutation networks* [21]. A Beneš network can perform any permutation π of *N* input items by passing them through O(*N* lg *N*) crossbar switches which operate on two items, either crossing them or passing them straight. The connections between the switches are fixed for a given *N*, only the cross-bar settings differ for different π.

This network is useful for our problem because (1) the SCOP can use cryptography to perform a cross-bar switch on two items resident on the host without the host learning which way the switch went, and (2) by doing this for all the switches in a Beneš network, the SCOP can permute the whole dataset without the host learning anything about the permutation, even though he observes all the record I/O. More specifically, to execute a switch the SCOP reads in the two records involved, internally crosses them or not, and writes them out encrypted under a new key so the host cannot tell if it was a cross or not. Since the network consists of 2 lg *N* columns of switches with *N/*2 switches each, and the SCOP can execute the switches column by column, he can use one key per column, thus never needing to store more than two keys at a time during the operation.

[4]The access pattern, ie. the sequence and values of I/O operations, will not be identical for all π, but must look identical to a computationally bound observer.

Networks similar to the Beneš are capable of performing other tasks obliviously, again making use of the fact that the SCOP can hide which way a unit operation (on two inputs) went, and by virtue of the fixed structure of the network, the ability to hide the setting of each unit extends to being able to hide the setting of the whole network. We later make use of *sorting* and *merging* networks in this manner.

1.2 Improvements to the Prototype

There are two areas where we saw the potential to improve our prototype: memory usage inside the SCOP, and the ability to update items privately.

Memory usage. Our prototype used two techniques which required $O(N \lg N)$ bits of storage inside the SCOP[5]. One was the storage of a permutation π selected uniformly at random from the set of all $N!$ permutations. The other was the execution of a Beneš network on the data items; in particular computing the switch settings of the network required $O(N \lg N)$ bits[6].

These "memory-hungry" techniques were not a problem for the kind of datasets we were treating, with $N < 2^{13}$ or so, and the memory available in the 4758. However even for $N = 2^{18}$, two objects *of* $N \lg N$ bits each would need more than 1MB, which begins to strain the 4758's memory. In any case, the memory requirements were, strictly speaking, inconsistent with the desire to have a small protected space.

Updates. Our prototype was really a Private Information *Retrieval* server, and did not have the option for clients to update the contents of data items. This ability could be of interest though, in more interactive applications of the PIR technique, for example if one wanted to build a private filesystem, which could be housed in a remote location but assure a user that nothing about his activities on the filesystem could be gleaned by the remote site.

2. RELATED WORK

Throughout this paper one notices references to Oblivious RAM (ORAM) [6]. This is because that problem has a very similar structure to hardware-assisted PIR, and the mechanisms developed there are for the most applicable here too. The ORAM problem is for a physically shielded but space-limited CPU to execute an (encrypted) program such that untrusted external RAM cannot learn anything about the program by observing the memory ac-

[5]Note that this is less than the $O(NM)$ storage which would be needed to hold the whole database: the size of data items we were working with was at least 1KB.

[6]It is not useful to store the settings on the host, as they are computed in an order dependent on the input, so an adversary could learn about π by observing this order

cess pattern. The CPU corresponds to the SCOP (acting on behalf of clients), and untrusted RAM corresponds to the host. The asymptotically slower solution presented there (square-root algorithm) is what we base our algorithm on.

The asymptotically superior ORAM solution (polylog algorithm), has a $O(\lg^4 N)$ per memory access overhead. An actual operation count reveals that it has a larger actual overhead that the square-root algorithm for about $N < 2^{20}$. Such large dataset sizes are practically infeasible for both algorithms on the hardware we currently have, so we have not experimented with the polylog algorithm.

The ORAM work has covered some of the aims we address in this paper, namely private reading and writing of memory words using a protected CPU with logarithmic in N memory size.

The new contributions over ORAM in this paper are:

- an asymptotically and practically more efficient method of re-shuffling the dataset between sessions (Section 3.2),
- a practically efficient session-transition scheme (Section 4.2),
- permutation using the Luby-Rackoff scheme (which has advantages, for example enabling us to compose and invert pseudo-random permutations) (Section 3.1),
- an actual implementation on commodity secure hardware.

Ostrovsky and Shoup introduced communication-efficient private information storage, the computationally secure version of which is based on the Oblivious RAM algorithm [14].

3. MEMORY USAGE

In this section we present solutions to the high memory needs of the previous prototype. As mentioned before, we had two distinct sources of super-logarithmic memory usage, both of which are addressed.

3.1 Permutation

We need a permutation on the set of integers $\{1,\ldots,N\}$. It should be storable in $O(\lg N)$ space, which rules out the use of a truly random permutation: it requires $O(N \lg N)$bits of storage. It should also be invertible, which is required by our re-shuffling algorithm (Section 3.2). Because of the storage restriction, we have to settle for a *pseudorandom* permutation, and the one we chose is the Luby-Rackoff-style cipher on $\lg N$-bit blocks, with 7 rounds (LR_n^7) [11].

An L-R cipher (on $2n$-bit blocks) is a *Feistel network* with independent pseudo-random round functions. A Feistel network consists of several iterated rounds $R_i(L, R) = (R, L \oplus f_i(R))$, where

- $L, R \in \{0,1\}^n$ are initialized such that $LR = x$, x being the plaintext,
- f_i are *round functions*, $f_i \in \{0,1\}^n \rightarrow \{0,1\}^n$. Note that they do not have to be permutations for the whole network to be a permutation—this is part of the point in fact, that non-invertible functions are used to produce a permutation.
- $\oplus$ is the bitwise XOR operation.

Luby and Rackoff initially proved chosen-plaintext security with 3 rounds, and chosen-ciphertext security with 4 rounds, in both cases with only a limited number of queries against the cipher oracle.

Recent results have improved the security bounds for higher-round L-R ciphers to state that LR^7_{2n} is indistinguishable from a truly random permutation by an unbounded adversary given m chosen-plaintext queries, where $m \ll 2^{n(1-\varepsilon)}$ [15]. The potential weakness to chosen plaintext attacks (CPA) is significant in our case because the host can mount such an attack by issuing requests to the SCOP (posing as a client), and observing which items in the shuffled dataset the SCOP accesses. In fact the host can harvest up to k chosen-plaintext pairs from the permutation π, where k is the number of retrievals in the session.

A variation on the basic L-R scheme has been conjectured to give a much higher resistance to CPA—unbalanced Feistel schemes which have round functions $f_i \in \{0,1\}^r \rightarrow \{0,1\}^{2n-r}$, where $r \neq n$. In particular Patarin conjectures that an unbalanced L-R scheme (as described, among others, by Naor and Reingold [13, Sect. 6]) on $2n$ bits, using round functions $f_i \in \{0,1\}^{2n-1} \rightarrow \{0,1\}$ (ie. boolean functions on $2n-1$ bits), are secure against CPA given m chosen-plaintext queries, where $m \ll 2^{2n(1-\varepsilon)}$ [15].

For the pseudo-random functions inside the cipher, we use TDES (which is hardware accelerated on the 4758) with expansion and compression to give a function on the required domain.

3.2 Shuffling the Dataset

Once we have established a random or pseudo-random permutation, we need to actually permute the records such that the server cannot learn anything about the permutation. As mentioned before, the Beneš network is not applicable if we are to use only logarithmic space. The algorithm to set its switches for a given permutation has resisted many simplification attempts.

The solution which we came up with takes advantage of the fact that only a small fraction of the dataset is touched during a query session. The untouched items do not need to be reshuffled, only the touched ones do. Informally, the procedure for reshuffling is as follows.

Let the current permutation be π_1. Let T be the touched items at the end of a session, and $\overline{T}$ be the remaining items, untouched. Let the size of T be

k (which is constant in our prototype, so as to limit the query response time). For the next session we generate a new permutation π_2. Also we assume that the indices of the items in T are available in a list L_T in the SCOP. Then we follow the following algorithm:

Reshuffling algorithm.

(i) Re-order the items in $\overline{T}$ so they are sorted by $\pi_2(i)$. We do not need to do this obliviously. We just need to hide what are the indices of T under π_2 (but not under π_1—this is already known).

(ii) Obliviously re-order the items in T so they are sorted by $\pi_2(i)$.

(iii) Obliviously merge the re-ordered T and $\overline{T}$, to give a dataset shuffled under π_2.

This yields savings both in time and space over using a Beneš network to do a full reshuffle. We will first describe in more detail the algorithms used, and then present the resources needed. We assume that we can compute inverse permutations, which is true with Luby-Rackoff style permutations. **Step (i)** is shown in Algorithm 1. **Step (ii)** can be directly performed using a *sorting network,* eg. one of Batcher's networks. However a more efficient approach is to use the Beneš network, after computing the permutation vector for the reordering needed. This can be accomplished using the list L_T, with one sorting step to obtain a sorted list of the indices in T under π_2[7]. **Step (iii)** can be performed using a merging network.

A good reference for sorting and merging networks is found in "Introduction to Algorithms" [4, chap. 27], and at the end of Section 1.1 we explain how such networks can be used to perform operations on a large dataset obliviously.

Notes. Step 6 in Algorithm 1 must take the same amount of time at every execution[8], but this is easy given the sorted array L_{T,π_2}, and takes constant time.

The initial shuffle. For the initial shuffle, which has to re-order all the items obliviously, we resort to the use of Batcher's bitonic sorting network. This method was used in the ORAM work for all shuffles of memory.

Sorting networks sort N items by passing them through a series of *comparators,* which are 2-input units that sort the two inputs. The connections between the comparators are fixed for a given N.

[7]Note that we had to perform this sorting at the start of Algorithm 1 too, so the output of that can be reused.

[8]Or an adversary could use timing attacks to deduce information about the indices of $\overline{T}$ under π_2.

Algorithm 1 Step (i) of the re-shuffle algorithm: Reordering the items in $\overline{T}$ from π_1 to π_2

```
Require: $T$ is the set of touched records, $\overline{T}$ are the remaining records.
Require: $D_{\pi_1}$: the whole dataset under $\pi_1$, on the host.
Require: $L_T$: list of the indices of $T$, in the SCOP.
 1: $L_{T,\pi_2} \leftarrow$ indices of $T$ under $\pi_2$, sorted                ▹ Using $L_T$
 2: $\overline{T}_{\pi_2} \leftarrow \emptyset$      ▹ The destination array (on the host) for the records in $\overline{T}$
 3: $j \leftarrow 0$                                          ▹ $j$ is an index under $\pi_2$
 4: while $j < N$ do
 5:     $j \leftarrow$ next index in $\overline{T}$ in order of $\pi_2$                ▹ guided by $L_{T,\pi_2}$
 6:     $r \leftarrow \pi_1(\pi_2^{-1}(j))$                              ▹ $r$ is an index under $\pi_1$
 7:     $R \leftarrow$ read_from_host $D_{\pi_1}[r]$
 8:     Tag $R$ with destination $j$          ▹ But this tag is hidden from the host
 9:     Append encrypted $R$ to $\overline{T}_{\pi_2}$               ▹ Recall $\overline{T}_{\pi_2}$ is on the host
10: end while
```

Step	Time cost	Space cost (in bits)
(i)	$O(k \lg k)$ for sorting, $O(N-k)$ for the loop	$O(k \lg N)$ for the indices of T
(ii)	$O(k \lg k)$ for the Beneš network	$O(k \lg N)$ for building and storing the permutation vector, $O(k \lg k)$ for the Beneš network
(iii)	$O(N \lg N)$ for the merging network	$O(\lg N)$ for indices

Table 1. Cost of the reshuffle algorithm. k is the number of queries in a session, same as the size of a touched set. Note that the cost of the merging network in the last step is the dominant one, and that is half the cost of a Beneš network on the same input size. Also the storage needed is $O(k \lg N)$, which is $O(\lg N)$ for constant k (which is how we set k). Even if $k = \sqrt{N}$ as in the ORAM square-root algorithm, the storage required is considerably sublinear.

Batcher's sorters have depth $\frac{\lg^2 N}{2}$, which is appreciably larger than the Beneš network which we have so far used, by a factor of $\frac{\lg N}{4}$, but since we only need to use it once, before the database can be used, this is not a big problem.

Our usage of the bitonic sorting network is very similar to how we used the Beneš network. First we tag each record r with its destination tag $d_r = \pi(r)$, and pass the records through the sorting network, with d_r as the key. We implement a comparator inside the SCOP such that the host cannot tell whether the two records were crossed or not.

4. UPDATES

The problem of evolving our previous design to support private updates of data items reduced to two main tasks: ensuring integrity of data, even against replay attacks[9]; and dealing with the fact that incoming updates render the data in any long-lived preprocessing steps stale: for example a shuffled dataset will be out-of-date by the time the shuffle is done (assuming that shuffles run in parallel to queries, which is necessary to avoid downtime between sessions).

The easy part was modifying the retrieval session to deal with (1) hiding whether a client request is an update or a retrieval, and (2) hiding which item in the working pool is being updated. The approach is to update *all* records in the working pool (but not all records in the dataset) with every request. In particular, for every record r in the pool, the SCOP writes either $\{r\}_K$, or $\{r_{new}\}_K$ if a new value r_{new} is provided by the client. The variable K is a new key generated for this encryption of the working pool only. Given this change of key, the host cannot tell if and where a new record was written. Note that the SCOP does not need to keep the keys for previous versions of the pool.

4.1 Integrity

The integrity of any object stored on the host is assured by first tagging it with a value t which specifies both the *physical* and *temporal*[10] location of the object, and then applying a keyed message authentication code (MAC) to the object and the tag. The location code and MAC are stored with the object on the host. For example, during the last step of a re-shuffle operation (the merging network) we have $t = \langle s, d, i \rangle$, where s is the current shuffle number, d is the depth within the network[11] (both temporal), and i is the item's current actual location in the dataset (physical). Thus, an adversary obviously cannot modify the item's contents undetected, but it also cannot substitute an item from an earlier time (ie. cannot perform a replay attack).

Of more interest is how to compute the temporal location of an object updated during a query session. Within the s^{th} query session, at the end of the i^{th} client request, the query SCOP has built up a working pool of touched records $P_s = \langle r_1, r_1, \ldots, r_i \rangle$. The temporal tag for each record in P_s would then be $t = \langle s, i \rangle$. The notable aspect here is that the SCOP can compute the temporal tag for each object which needs it while maintaining only a fixed small amount of state—s and i in this case. This temporal tagging with small state is the same notion as the "time-labelled" property expounded for some of the Oblivious RAM simulations, and also used to protect against tampering and replay attacks [6].

[9]Replay attacks are where the adversary replaces an item with another one which has a correct checksum/MAC, but comes from a previous execution of the algorithm.

[10]"Temporal" in the sense of where in the timeline of the algorithm the object is located.

[11]The merging network has $\frac{1}{2}\lg N$ levels of $N/2$ independent comparators each, and the depth is the current level number during an execution of the network.

4.2 Session Continuity

The problem of transitioning between query sessions is trivial in the case of read-only PIR: since the database contents are assumed static, several shuffled copies can be produced in advance and used immediately whenever needed—the shuffle data does not go stale. If updates are supported though, pre-shuffling is not an option as the shuffled datasets *will* be stale soon. Even worse, updates will occur between the start and end of a shuffle, requiring them to be incorporated into the output of that shuffle before it can be used. Here we describe our scheme for transitioning between sessions.

Given that we run a shuffle concurrently with a query session, the output of the shuffle will not contain the updates received during that session. We deal with this problem by incorporating the records T_i touched during session i into the working pool of session $i + 1$ from the beginning. This means that session $i + 1$ will touch each record in T_i at every operation, in addition to its own accumulating T_{i+1}. At the end of session $i + 1$, its working pool will contain both T_i and T_{i+1}.

Overview of the algorithm. In Figure 2 we show a diagrammatic representation of the actions of all the components during one full session.

We first note that the working pool of the query session consists of two parts—the set of records touched during that session (which is empty at first), and the ones touched during the last session.

At the end of session $i - 1$, the query SCOP has produced a new touched set T_{i-1}, and the shuffler has produced a new shuffled dataset D_{π_i}. The new session i starts by writing the items in T_{i-1} into D_{π_i} (directly, one by one), and also adding them to its working pool.

The shuffler begins to re-shuffle the dataset D_{π_i}, with touched set T_{i-1}, for use in session $i + 1$ (recall from Section 3.2 that we only need to obliviously reorder the items in a shuffled dataset which have been touched since the last shuffle—these items are now T_{i-1}).

5. EXPERIMENTAL RESULTS

Here we present some performance results from our prototype, whose constitution is described in Section 1.1.

In Figure 3 is shown the running time for the reshuffle operation described in Section 3.2. In Figure 4 we show how long it takes the query SCOP to process queries. Putting these two measurements together gives an idea of what kind of service this prototype can offer. In Table 2 we show the query processing time possible for different N, with two limiting factors: the query processor speed, and the re-shuffle speed (keeping in mind that the query processor can

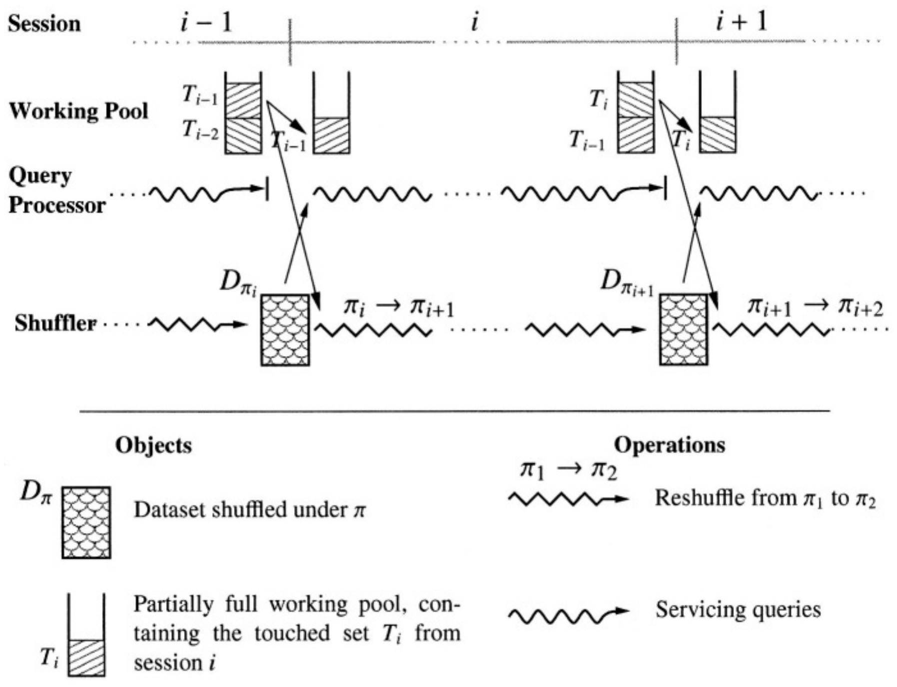
Session
$i-1$
i
$i+1$
Working Pool
T_{i-1}
T_{i-2}
T_{i-1}
T_i
T_{i-1}
T_i
Query Processor
D_{π_i}
$D_{\pi_{i+1}}$
Shuffler
$\pi_i \rightarrow \pi_{i+1}$
$\pi_{i+1} \rightarrow \pi_{i+2}$
Objects
Operations
D_π
Dataset shuffled under π
$\pi_1 \rightarrow \pi_2$
Reshuffle from π_1 to π_2
Partially full working pool, containing the touched set T_i from session i
T_i
Servicing queries

Figure 2. A snapshot of the overall algorithm across one session. At the end of session $i-1$ the shuffler has prepared $\boldsymbol{D_{\pi_i}}$, and the query SCOP has a working pool containing the touched sets $\boldsymbol{T_{i-1}}$ and $\boldsymbol{T_{i-2}}$ (see Section 3.2). Then, the shuffler begins a reshuffle with permutation $\boldsymbol{\pi_{i+1}}$ and with touched set $\boldsymbol{T_{i-1}}$. The query SCOP begins a new session i with $\boldsymbol{T_{i-1}}$ in its working pool, and filling in its own $\boldsymbol{T_i}$.

only service $k = 128$ queries before needing a new shuffled dataset from the shuffler).

6. FUTURE WORK AND CONCLUSIONS

We have presented the evolution of our previous work on a hardware-assisted private information retrieval prototype—improved performance in terms of both running time and space, and the ability to update items privately. The prototype gives reasonable performance on dataset sizes up to about 10,000, and can benefit easily from parallelism via extra hardware units.

There are several avenues of interesting and useful further investigations.

We did our prototype work on the IBM 4758, but alternate trusted hardware is emerging. We are particularly interested in exploring the hardened-CPU variations (e.g., [10, 12, 20]), since these devices may provide higher performance, as well as being cheaper and more ubiquitous.

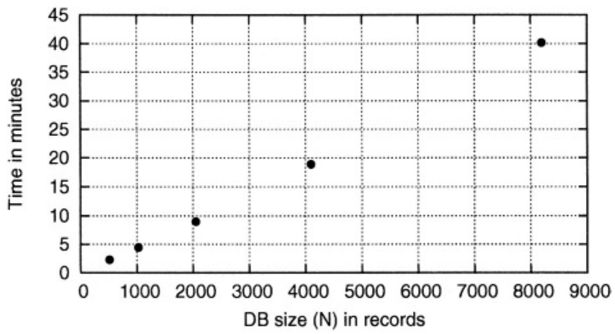

Figure 3. Duration of re-shuffling, for varying dataset sizes. The record size was 850 bytes in all cases. The dominant operation is the oblivious merge, all the others take much less time.

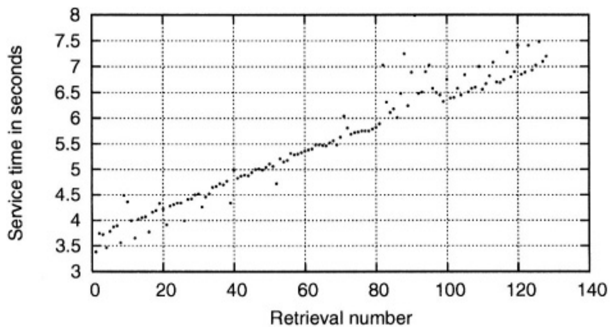

Figure 4. How long does the query SCOP take to service a request? Here are the times during one query session. Recall that the working pool starts with all the items touched during the previous session, in this case 128. The pool accumulates 128 more during the session.

Experiments with the poly-logarithmic oblivious RAM scheme by Ostrovsky [6] could be interesting, for both PIR on larger datasets, and oblivious program execution. Especially now that hardened CPU's, similar to the model used for ORAM, are coming into the picture, the tool of running arbitrary programs obliviously may be practically useful.

As mentioned earlier, private information storage could be a useful primitive for a strongly privacy-protected remote filesystem, providing the "block device" on top of which a filesystem could be built. Relevant here is work analyzing the applicability of block-PIR protocols such as we have described to retrieval of linked structures, eg. web pages [8].

Acknowledgments. This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), Internet2/AT&T and the Office for Domestic Preparedness, De-

N	Query processor limit	Shuffler limit	Response Time
1024	5.5	2.0	5.5
2048	5.5	4.1	5.5
4096	5.5	8.7	8.7
8192	5.5	18.5	18.5

Table 2. **Query response times (in seconds) attainable with different sizes of datasets. The two limit columns show how the respective operations limit the response—the query processor with its average latency (from Figure 4), and the shuffler by virtue of having to complete a whole re-shuffle before the next session can begin. In the $N \geq 4096$ cases, the query SCOP could handle more hits, but a single shuffler is not producing shuffled datasets quickly enough. An easy way out here is to do the merge step of the re-shuffle in parallel, using two or more SCOPs, and gaining linear speedup with the number of SCOPs, as the merging network is actually intended for parallel use. For $N = 1024$, the query SCOP can be always busy and the shuffler will keep up.**

partment of Homeland Security (2000-DT-CX-K001). We also thank IBM Research for their assistance with the 4758 secure coprocessors.

This paper does not necessarily reflect the views of the sponsors.

The authors are grateful to Dmitri Asonov, Dan Boneh, Rafail Ostrovsky and the anonymous referees for their helpful comments and discussion.

References

[1] Dmitri Asonov and Johann-Christoph Freytag. Almost optimal private information retrieval. In Dingledine and Syverson [5], pages 209–223. LNCS 2482.

[2] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Eurocrypt 1999,* Prague, Czech Republic. Springer Verlag. LNCS 1592.

[3] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM,* 45:965–982, 1998.

[4] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms.* McGraw-Hill, second edition, 2001.

[5] R. Dingledine and P. Syverson, editors. *Privacy Enhancing Technologies,* San Francisco, CA, April 2002. Springer. LNCS 2482.

[6] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM,* 43(3):431–473, 1996.

[7] Alex Iliev and Sean Smith. Privacy-enhanced directory services. In *2nd Annual PKI Research Workshop,* Gaithersburg, MD, April 2003. NIST.

[8] Dogan Kesdogan, Mark Borning, and Michael Schmeink. Unobservable surfing on the world wide web: is private information retrieval an alternative to the MIX based approach? In Dingledine and Syverson [5]. LNCS 2482.

[9] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science,* pages 364–373, 1997.

[10] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the*

9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 168–177, November 2000.

[11] M. Luby and C. Rackoff. How to construct pseudo-random permutations from pseudo-random functions. *SIAM Journal on Computing,* 17(2):373–386, 1988.

[12] P. McGregor and R. Lee. Virtual Secure Co-Processing on General-purpose Processors. Technical Report CE-L2002-003, Princeton University, November 2002.

[13] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology,* 12(1):29–66, 1999.

[14] Rafail Ostrovsky and Victor Shoup. Private information storage. In *ACM Symposium on Theory of Computing.* ACM, 1997.

[15] Jacques Patarin. Luby-Rackoff: 7 rounds are enough for $2^{n(1-\epsilon)}$ security. In *Advances in Cryptology–CRYPTO 2003,* pages 513–529. Springer-Verlag, Oct 2003.

[16] Sean Smith. Outbound authentication for programmable secure coprocessors. In *7th European Symposium on Research in Computer Science,* Oct 2002.

[17] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks,* 31:831–860, 1999.

[18] S.W. Smith and D. Safford. Practical server privacy using secure coprocessors. *IBM Systems Journal,* 40(3), 2001. (Special Issue on End-to-End Security).

[19] National Institute Of Standards and Technology. Security requirements for cryptographic modules.
`http://csrc.nist.gov/publications/fips/fips140-1/fips1401.htm`, Jan 1994. FIPS PUB 140-1.

[20] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing,* pages 160–171. ACM Press, 2003.

[21] Abraham Waksman. A permutation network. *Journal of the ACM,* 15(1):159–163, Jan 1968.

[22] Bennet S. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.