PRIVATE INFORMATION STORAGE WITH LOGARITHMIC-SPACE SECURE HARDWARE

Alexander Iliev[1]
sasho@cs.dartmouth.edu

Sean Smith[1]
sws@cs.dartmouth.edu

[1] *Department of Computer Science*
Dartmouth College

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 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, 10].

The other track attempts to produce *Practical* PIR schemes [8, 20–1] 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) [19].

1.1 Existing Prototype

Our previous work on Practical PIR (PPIR) [8] produced a PPIR prototype running on the IBM 4758 secure coprocessor with Linux [19], 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 [24]. It also includes mechanisms to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [18]. 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

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

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 scrutiny currently offered—FIPS 140-1 level 4 [21]. 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, which we sometimes refer to as K. If this space was large enough to hold the whole dataset, the problem would be solved, as clients could negotiate a secure session with K, and then retrieve their data. Since K 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 [7], 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.

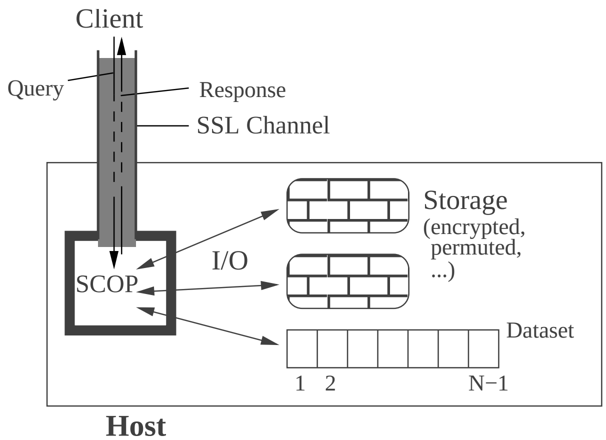

Figure 1. The setup of hardware assisted PIR

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 the identity of *single* retrievals, as then an attacker could learn the popularity of individual items, and correspondence between requests, eg. "Agnes 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, [20] kept the dataset unprocessed on the host. 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 has been developed by Goldreich and Ostrovsky for the Oblivious RAM problem [7]. 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:

[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.

Randomly permuting the contents of records 1 through N. First, K encrypts each record in the dataset. Then, K (pseudo)randomly selects a permutation π of $[1..N]$, and relocates the contents of each record r, $1 \le r \le N$, 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 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 (π)[4]. [7].

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 size k, which is scanned in its entirety for every retrieval. For efficiency we actually only scan the "active" items in P_S, thus the first retrievals are serviced faster (as is seen in Figure 4). 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* [23]. 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

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

[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.

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.

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 and storing the switch settings of the network both required $O(N \lg N)$ bits.

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.

[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.

2. Related Work

Throughout this paper one notices references to Oblivious RAM (ORAM) [7]. This is because that problem has a very similar structure to 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 access pattern. The asymptotically slower solution presented there (square-root algorithm) is what we base our algorithm on.

The asymptotically superior 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:

- Application of the techniques to the PIRW problem,
- an asymptotically and practically more efficient method of re-shuffling the dataset between sessions,
- permutation using the Luby-Rackoff scheme (which has advantages, for example enabling us to compose and invert pseudo-random permutations),
- an actual implementation on commodity secure hardware.

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. We have to settle for a *pseudorandom* permutation, and the one we chose is the Luby-Rackoff-style cipher on n-bit blocks, where $n = \lg N$, with 7 rounds (LR_n^7) [12].

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 $m \ll 2^{n(1-\varepsilon)}$ [16]. The potential weakness to chosen plaintext attacks 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. Since k is likely to be in the region of 2^8 in our case, it is actually greater than the range covered by the above proof of security, for $N < 2^{16}$ at least. We proceed without the proven assurance of a cipher secure against an all-powerful adversary, but hopeful that a computationally bounded adversary will have sufficiently low advantage in distinguishing this cipher from a truly random one.

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 ($\{0,1\}^n$).

Note that the L-R scheme allows the SCOP to calculate inverse permutations and compositions as well.

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 retrieval 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 (ie. the working pool) at the end of a shuffle, and T' be the remaining items, untouched. Let the size of T be K. 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:

1 Re-order the items in 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).

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

3 Obliviously merge the re-ordered T and 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 one** is shown in Algorithm 1. **Step two** 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[6]. **Step three** can be performed using a merging network.

A good reference for sorting and merging networks is found in "Introduction to Algorithms" [5, 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, 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. After some searching in the permutation network literature (eg. [15]), our conclusion is that it is the simplest and probably the most efficient mechanism for performing an arbitrary permutation using only logarithmic space. This appears to have also been the conclusion of the Oblivious RAM authors as they use Batcher's sorters to shuffle memory.

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

Algorithm 1 Step 1: Reorder the items in T' from π_1 to π_2

Require: D_{π_1}: the whole dataset under π_1, on the host.
Require: L_T: list of the indices of T, in the SCOP.

```
L_{T,π2} ← indices of T under π2                          ▷ Using L_T
Sort L_{T,π2}
T'_{π2} ← ∅        ▷ The destination array (on the host) for the records in T'
j ← 0                                                    ▷ j is an index under π2
while j < N do
    j ← next index, under π2, in T'                      ▷ guided by L_{T,π2}
    r ← π1(π2^{-1}(j))                                   ▷ r is an index under π1
    R ← read_from_host D_{π1}[r]
    Tag R with destination j              ▷ But this tag is hidden from the host
    Append encrypted R to T'_{π2}                    ▷ Recall T'_{π2} is on the host
end while
```

Step	Time cost	Space cost (in bits)
1	$O(K \lg K)$ for sorting, $O(N - K)$ for the loop	$O(K \lg N)$ for the indices of T
2	$O(K \lg K)$ for the Beneš network	$O(K \lg N)$ for building and storing the permutation vector, $O(K \lg K)$ for the Beneš network
3	$O(N \lg N)$ for the merging network	$O(\lg N)$ for indices

Table 1. Cost of the reshuffle algorithm. 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.

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.

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[7]; 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 retrievals, 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* [8] 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[9] (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 retrieval session. Within the s^{th} retrieval session, at the end of the i^{th} client request, the retriever 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

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

[8] "Temporal" in the sense of where in the timeline of the algorithm the object is located—as we discuss in the sequel.

[9] 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.

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 [7].

By way of contrast, if records were updated individually, without re-updating all the records in the current working pool, the SCOP would have to keep track of the update count of each record in order to compute a temporal tag for it. Other options in that case include:

- **Hash trees**: a tree of cryptographic hashes with every node being the hash of all its children, and the leaves being the actual data items [14]. A hash tree can enable the SCOP to verify any of N objects while keeping only a single hash internally, at $O(\lg N)$ time expense per verification and update.

- **Incremental multiset hashes**: collision resistant hashes of multisets, as opposed to ordered sequences. These can be used to verify *offline* a series of M read/writes to untrusted storage with $O(M)$ time expense [4].

4.2 Session Continuity

The problem of transitioning between retrieval 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.

Overview of the algorithm. In Figure 2 we show the outline view of one iteration of the whole algorithm. Essentially there are several "communicating processes":

- **Shuffler** is always shuffling a dataset, starting with a fresh dataset each time, which is stale by the time the shuffle is done.

- **Helper** thus has to prepare a set of updated records for oblivious merging with a shuffled dataset.

- **Retriever** is serving requests to read or update items, while building up its working pool of records. A session is split into two parts for the purposes of propagating updates to the shuffler. In the first, the retriever collects an update pool of records P_A. These are passed to the helper to prepare for merging with the shuffle-in-progress. In the second part, the retriever collects P_B. These updates are applied to the dataset D_{π_1}

by writing them in directly[10], and also added to the second session's working pool. This step will reveal the correspondence between π_0 and π_1 on the records in P_B, but this is not a problem because the items for which the correspondence is revealed become part of the working set.

These processes would ideally run in parallel, on separate SCOPs, which is the setup we use. They could just be concurrent on a single processor though.

5. Experimental Results

Here we present some performance results from our prototype.

In Figure 3 are shown running times for the lengthy operations of the algorithm: shuffling the dataset and preparing update pools for merging. In Figure 4 we show how long it takes the retriever to process queries. Putting these two measurements together gives an idea of what kind of service this prototype can offer. First we note that we set the size of P_A (see Figure 2) to 256 elements. This choice ensures the reponse time of the retriever is less than 10 seconds. Then the size of P_B is decided by how long it takes to prepare P_A for merging, so that the expansion, the shuffle and the retrieval session finish at the same time. In Table 2 we give a rough idea of what performance the prototype as a whole can offer.

N	$\lvert P_B \rvert$	Bottleneck process	Max query delay
1024	40	retriever	10
2048	70	even	10
4096	130	shuffler	20

Table 2. Rough estimates of query delays (in seconds between hits) attainable with different sizes of datasets. (That is, for $N = 1024$, we can sustain a query rate of one every 10 seconds, with each response sent before the next query arrives.) In the $N = 4096$ case, the retriever could handle more hits, but a single shuffler is not producing shuffled datasets quickly enough. An easy way out here is to do the shuffle in parallel, using two or more SCOP's. For $N = 1024$, the retriever can be always busy and the shuffler will keep up.

6. Future Work and Conclusions

There are several avenues of interesting and useful further investigations.

One is to try to devise a memory efficient way to compute and execute a pseudorandom permutation obliviously in reasonable $O(N \lg N)$ time (ie. without using the AKS sorting network). One area which we explored is to use a pseudorandom function $f : [0..S_{B_N}] \to \{0, 1\}$, where S_{B_N} is the number

[10] ie. for each $r \in P_B$, the retriever updates slot $\pi_1(r)$ in D_{π_1} with the contents of r

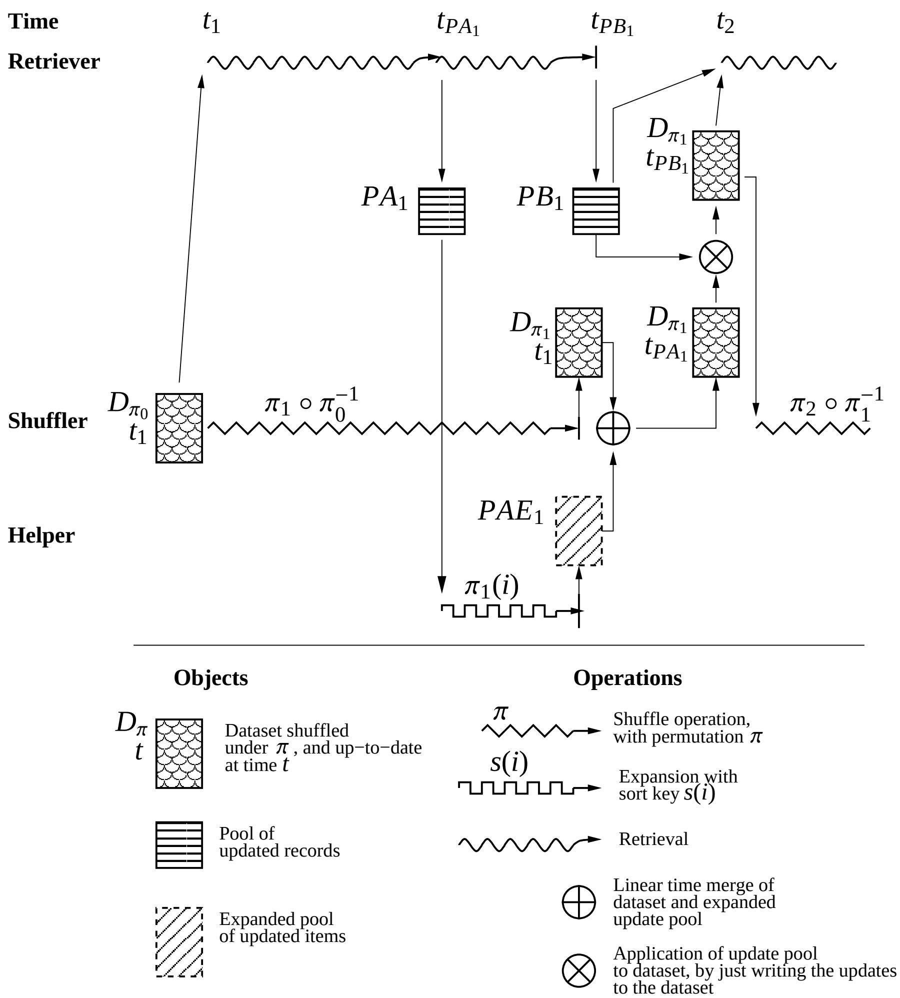

Figure 2. A snapshot of the overall algorithm across one session. We start with a pre-shuffled dataset D_{π_0}. Each subsequent shuffle i uses as a permutation $\pi_i \circ \pi_{i-1}^{-1}$, in order to move the dataset to the next permutation π_i directly. (Recall that the SCOP can calculate inverses permutation maps and compose permutation maps.) The end of the retrieval session comes when update pool PB_1 is needed to be applied to the shuffled dataset in step $\bigotimes$.

of switches in a Beneš network with N inputs, to construct a pseudorandom Beneš network, which can then be used for the two tasks. Such a construct has been considered before as a way to build a block cipher [17], but the apparent strength against chosen plaintext attacks seemed insufficient for the small block sizes we are interested in. We have not written off the possibility that a variant of this idea may prove fruitful.

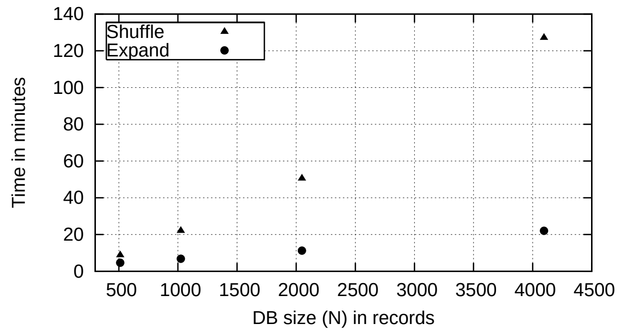

Figure 3. Duration of shuffling and obliviously merging updates, for varying dataset sizes. The record size was 850 bytes in all cases.

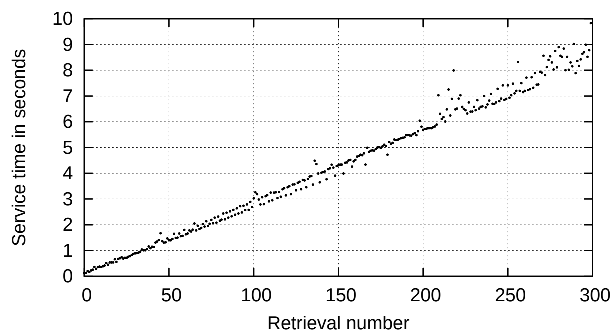

Figure 4. How long does the retriever take to service a request? It increases throughout the retrieval session—this plot shows the duration of each of 300 retrievals in one session.

Another avenue is to try out in practice the poly-logarithmic oblivious RAM scheme by Ostrovsky [7], for both PIR 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.

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., [22, 11, 13]), since these devices may provide higher performance, as well as being cheaper and more ubiquitous.

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 [9].

References

[1] Dmitri Asonov and Johann-Christoph Freytag. Almost optimal private information retrieval. In Dingledine and Syverson [6], 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] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003*, volume 9 of *LNCS*, pages 188 – 207. Springer-Verlag, 2003.

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

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

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

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

[9] 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 [6]. LNCS 2482.

[10] 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.

[11] 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.

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

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

[14] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology–Crypto '89*, LNCS, pages 218–238. Springer Verlag, 1990. Introduces hash trees. Originally a manuscript from 1979.

[15] A. Yavuz Oruç. Multiple tracks of research on interconnection networks. In *1995 International Conference on Parallel Processing Workshop*, pages 16–23, 1995.

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

[17] Michael Portz. On the use of interconnection networks in cryptography. In D.W. Davies, editor, *Advances in Cryptology–EUROCRYPT '91*, LNCS 547, pages 302–315. Springer-Verlag, 1991.

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

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

[20] 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).

[21] 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.

[22] 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.

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

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