Dartmouth College
Dartmouth Digital Commons

Computer Science Technical Reports | Computer Science

2-11-2003

Privacy-enhanced credential services

Alex Iliev
Dartmouth College

Sean Smith
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/cs_tr

Part of the Computer Sciences Commons

Dartmouth Digital Commons Citation
Iliev, Alex and Smith, Sean, "Privacy-enhanced credential services" (2003). Computer Science Technical Report TR2003-442. https://digitalcommons.dartmouth.edu/cs_tr/208

This Technical Report is brought to you for free and open access by the Computer Science at Dartmouth Digital Commons. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Privacy-enhanced credential services

Alex Iliev
sasho@cs.dartmouth.edu

Sean Smith
sws@cs.dartmouth.edu

DRAFT of February 11, 2003

Dartmouth Computer Science Department Technical Report TR2003-442

Abstract

The use of credential directories in PKI and authorization systems such as Shibboleth introduces a new privacy risk: an insider at the directory can learn much about otherwise protected interactions by observing who makes queries, and what they ask for. Recent advances in Practical Private Information Retrieval provide promising countermeasures. In this paper, we extend this technology to solve this new privacy problem, and present a design and preliminary prototype for a LDAP-based credential service that can prevent even an insider from learning anything more than the fact a query was made. Our preliminary performance analysis suggests that the complete prototype may be sufficiently robust for academic enterprise settings.

1 Introduction

In this paper, we identify a privacy risk in PKI and other organization-centered authorization systems; we also offer a design (and partial prototype) of a practical solution.

Hippocrates advised physicians to "first, do no harm." We would also like to apply this dictum to the design and deployment of new security infrastructure. While examining whether new technology solves existing security problems, we should also ask: does it create new ones?

Traditional hierarchical PKI (as well as other authorization schemes) can potentially solve many problems regarding interaction within and across organizational boundaries. However, an artifact of this focus on organization is *centralization*. Advanced cryptography and protocols provide security and privacy as an organization's members interact with each other and the world. But making this all work typically requires the organization to maintain a centralized credential server, that offers the right tokens and certificates to the participants when they need them.

Although the cryptography they enable can solve security and privacy problems, the existence of these servers creates new ones: because many interaction with user A require queries to the credential server, it is possible for the credential server to learn a great deal about these interactions, by monitoring who is asking for which credentials. Securing interaction against outside adversaries thus has the unwanted side-effect of enabling attacks on privacy by organizational insiders.

Section 1.1 and Section 1.2 will consider some immediate manifestations of this problem. Section 2 outlines the technology that we and others have helped produce, that may address these issues. Section 3 then explains our design and (not yet complete) prototype that applies this technology to solve this problem. Section 4 and Section 5 examine whether this design will perform well in practice. Section 6 concludes with some directions for future work.

1.1 Certificate Directories

In the standard[1] approach to PKI, hierarchies of CAs and users emerge that mirror organizational hierarchies.

Public-key interaction requires knowing the public key of the other party, and being able to bind this keyholder to some relevant real-world property (such as identity or role). Within a user population served by a single root, a *public key certificate* provides this information; in more complex hierarchies, a multi-step *path* of certificates may be necessary.

To provide these certificates, organizations set up directories (typically via LDAP).

Within a population, if Alice wants to send a secret message to Bob, she needs to obtain Bob's public key. Typically, she asks a directory for this. If Bob receives a message from Alice and wants to verify a signature, he needs her certificate. If Alice did not provide this with the message, then Bob needs to ask the directory; even if Alice did provide it, Bob may wish to check if it's still valid.

Across different populations, parties may need to ask directories for additional certificates to construct trust paths. In more general settings, such as trust decisions based on attribute certificates as well as identity certificates, additional directory queries may be involved.

Consider the privacy implications if the adversary Mallory operates the directory. Alice may be using encryption on her message because she wants to keep this secret. But because she needs Bob's certificate, Mallory knows that Alice is sending a message to Bob. If Bob needs to obtain Alice's certificate (or check whether it's valid), he must ask the directory, so Mallory knows that too. If Alice and Bob take the precautions of using protected channels for their message, Mallory still learns of the interaction via the PKI at the end points. Even if Alice and Bob did not take precautions, the use of PKI lowers the work required for Mallory—rather than monitoring network traffic, she can just log queries to a directory.

1.2 Shibboleth

Shibboleth is a developing system to provide user authorization for access to resources in remote sites [8]. The user is assumed to have a *home* site, which can provide information about her. The resources are located at the *target* site. The simplified procedure, illustrated in Figure 1, is that the SHIRE[2] at the target site establishes an opaque *handle* for a user, after which the SHAR[3] uses this handle to request user attributes from the AA[4] at the home site. The attributes are then used to make an authorization decision.

Because the user handle is opaque to the target site components, they do not learn anything about the user beyond the attributes given by the AA. This is the main reason why Shibboleth claims to be privacy sensitive. What is not covered though, is that the home site can learn a lot about their users' online activities—which target sites they visit, and in some cases even the exact URL's.

For example, say John from Dartmoor College occasionally needs access to `http://webofscience.com/LegalizeIt`, `salon.com/archives/palestine/`, and `pop-music-journal.com/sexpistols/`, and these sites require Shibboleth authorization for users of subscribing institutions[5]. The SHAR at each web site will ask the AA of Dartmoor for some attributes of John, by passing John's opaque session handle (opaque to the sites, not to the AA), and the URL being requested. The URL is

[1] We note that dissent exists.
[2] Shibboleth Indexical Reference Establisher
[3] Shibboleth Attribute Requester
[4] Attribute Authority
[5] Perhaps the sites also offer pricey personal subscriptions.

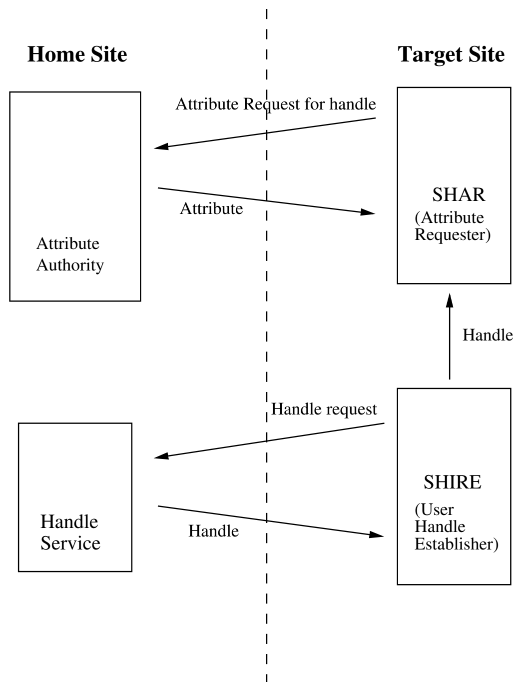

Figure 1: Shibboleth

passed to the AA so it can decide which attributes to release, based on Attribute Release Policies. The attributes will likely be non-identifying, like confirmation of institution membership, or age, so the sites do not receive any personal information about John. The AA at Dartmoor however does see which sites (including URLs) are asking for attributes for John, and if Mallory at the AA wished to do so, she could log this information.

2 Background

In this section, we examine some relevant technology which can enable a solution to the server privacy problem as described above. We note that work by Stefan Brands offers a much more general solution to privacy concerns in current PKI [3, 4]. Our solutions here are more incremental and have the potential to be deployed until such time as a more general overhaul of PKI is in place.

2.1 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 [16]. It also includes mechanisms to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [10]. The coprocessor is attached to a *host* computer. Since the secure coprocessor we use is implemented as a PCI card, we sometimes refer to a secure coprocessor as a *card*.

2.2 Private Information Retrieval

The problem of *private information retrieval* considers how a user can obtain a particular record from a large set a server offers, without the server learning anything about which record was requested. Simply encrypting the records does not solve the problem; the server can still learn popularity of individual records, correspondence between requests, and (if the server colludes with a user) can learn what any given record decrypts to.

Theoretical computer scientists developed many algorithms (e.g., [6, 5]) through which users, servers, and sometimes other parties could carry out computation and achieve PIR.

2.3 Practical PIR

Smith and Safford [12] then proposed the problem of *practical* PIR: using existing systems, can we provide PIR along the lines of a Web model: the user establishes a shared key, issues a request, waits a short while, then receives the response?

Their solution used COTS[6] secure coprocessors and assumed that a coprocessor can only hold a fixed small number of records internally at one time. Their scheme consists of handling a query to a PIR server by having a secure coprocessor read sequentially through *all* the records in the database (which is kept on the host), keep the correct record internally and return it to the user. The running time of a query is linear in the database size.

Asonov et al [2] then improved the Smith-Safford scheme by decreasing the processing time for a query at the expense of a periodic preprocessing step. We elaborate on this scheme in the next section.

2.4 Asonov's Scheme

The setup for this scheme is that the database consists of N records, numbered from 0 to N-1. They may or may not be originally encrypted, depending on whether the contents need to be kept private. The records are stored on the host, and accessed from the secure coprocessor (the *card*) via a simple API:

- `Record-text read_record(position)` and
- `write_record(Record-text, position)`.

The Record text may be encrypted for reads, and will be encrypted and MAC'ed for writes. An assumption is that at least two records can be stored on the card at a time.

The scheme is divided into two parts:

1. Preprocessing, where the database is shuffled such that the host has no information about the positions of records in the shuffled version.
2. Retrieval, where the card fetches records from the shuffled database.

2.4.1 Preprocessing

Shuffling is done in O(N^2) time, as follows. A uniform random shuffle vector `V` is generated such that record number `V[i]` goes to position `i` of the shuffled database. Then, for each postion `i`, the

[6]Commercial Off The Shelf

card sequentially reads *every* record from the host, keeps record number `V[i]` internally, and writes it out to position `i` of the shuffled database. The host does not know what `V[i]` is, so does not learn anything about the record going into shuffled position `i`.

2.4.2 Retrieval

For the first record retrieved after a shuffle, the card simply gets the record's shuffled position from its shuffle vector `V`, and retrieves that position. For the nth retrieval (call it record R_n) after a shuffle, the card re-fetches *all* $n-1$ records previously retrieved, then fetches and returns R_n. If R_n was in the n-1 already retrieved records, it is kept internally to be returned, and the nth record fetched is a random one not previously touched.

3 Our Extensions and Prototype

We can solve the privacy problem for credential servers by using a PIR server for the information source (e.g, the Shibboleth AA, or a certificate directory). Users could then have assurance that the system operator is not observing their queries [7].

The question, then, is can we build a credential server using PPIR technology and COTS hardware that performs reasonably well?

We decided that the outside interface should be LDAP[9], currently the most popular directory access protocol; we will make the limiting assumption that the querier can only specify one fully named record. This limitation is not unreasonable—asking a directory for the certificates of all Bobs does not seem like an indispensable operation.

We then consider the issues: Section 3.1 discusses extending PPIR to deal with *named* records; Section 3.2 presents a new approach to the shuffling step that decreases the time from $O(N^2)$ to $O(N \log N)$; Section 3.3 discusses our PPIR setup; Section 3.4 discusses our prototype implementation; and Section 3.5 discusses the overall credential server architecture.

3.1 Named Records via Hashing

Real database records are usually named as opposed to numbered. The approach we took to dealing with names in this prototype was to implement the database using hashing with chaining. Thus, hashing a record name yields a bucket number, and inside this bucket is the needed record. We effectively ran the Asonov scheme with numbered buckets. Within a given bucket, a record was retrieved by reading in all the records sequentially and keeping the right one.

The hash function we used is due to Dan Bernstein and was chosen as it was simple, seemed well-recommended, and performed well compared to several others we tried. The most important metric we used was the size of the largest bucket produced. The function is, for a string `str[0..n-1]`:

```
hash(str[0..i]) = hash(str[0..i-1]) * 33 ^ str[i].
```

Some other more complicated name resolution options we considered are

[7]For Shibboleth, further steps needed would be to make the HS[8] privacy-protected too, or a reasonable guess could be made at the identity of a request for attributes which comes soon after a login at the HS.

[9]Lightweight Directory Access Protocol

1. We could hold a data structure of all the record names inside the card, possibly with some index for fast searching, and use this to look up numbers from names. If we assume names to be 20 bytes on average, for 10,000 records this structure would be 200K memory minimum, which could possibly be kept inside the card. Also possible would be to outsource the name resolution to another card. This approach could also be useful with providing substring name matching.

2. We could use a *perfect hash function* [9]. This needs to be constructed especially for the current set of names, but then hashes each name into a unique bucket. Perfect hash functions tend to come with an index whose size is comparable to the name set's size, but since this index would consist of about N integers (sized 2 bytes each), it would certainly be a lot smaller than the full name table.

3.2 Private Shuffling with Permutation Networks

We have planned an alternative and faster shuffling algorithm which we shall sketch out here. It is based on a *Permutation network* - a network of *switches* wired together in a fixed manner and intended to perform a given permutation of its inputs [15]. A switch has two inputs and two outputs, and it may cross the inputs, or pass them on straight. By propagating values along the wires and through the appropriately set switches, any permutation of the input can be produced at the output. An example 4-input network is shown in Figure 2. A permutation network for N inputs can be built recursively as shown in Figure 3. [7] This network clearly consists of $\Theta(N \log N)$ switches. Setting all the switches to achieve a given permutation is possible with a $\Theta(N \log N)$ algorithm [14, 1].

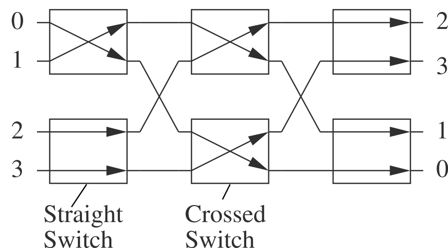

Figure 2: A Permutation network with 4 inputs, performing the permutation $\langle 2, 3, 1, 0 \rangle$

Permutation networks have in fact been proposed for use in a related field—mix networks for anonymizing email [1]. The similarity lies in the shared goal of erasing any observable relationship between the inputs and outputs of a shuffle or mix net.

A switch can be interpreted for our shuffling scenario as follows. The coprocessor reads in two records (the inputs), possibly switches their places, and writes them out to the same two positions. The host should be unable to tell if the two records were switched or not. This can be achieved by reencrypting the records with new keys for example.

Finally, a shuffle using such a permutation network would consist of the card internally generating a random permutation, generating a network for its chosen permutation, and then executing all the switches in order (column-major order looks sensible). Generating the network takes $\Theta(N \log N)$ time, as does executing the switches.

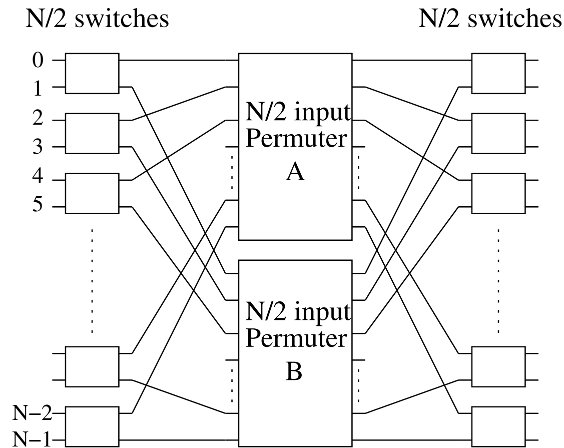

Figure 3: A Permutation network for N inputs built from N switches and 2 networks with N/2 inputs each. Each of the switches on the left have one output wired to permuter A, and one output to B. The switches on the right have their inputs similarly connected. This construction is straighforward but not entirely minimal—N/2 switches can be removed while still enabling any permutation to be executed [15].

3.3 System Setup

Our prototype runs in the IBM 4758 secure coprocessor with Linux [11]. 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 [13]. It has an Intel 486 processor, 4MB of RAM and 4MB of FLASH memory. It also has cryptographic acceleration hardware. It connects to its host via PCI. Our host runs Debian Linux, with kernel version 2.4.2-2 from Redhat 7.1 as needed by the 4758/Linux device driver.

Linux is an experimental operating system for the 4758, which runs CPQ/++ in production, but 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 card with Linux.

3.4 PPIR Implementation

An overview of the components of our PPIR prototype is shown in Figure 4. Our implementation of retrieval with hashing is illustrated in Figure 5. Shuffling in this prototype is a straightforward implementation of the naive algorithm in Section 2.4.1. We make our code available at `http://www.cs.dartmouth.edu/~sasho/privdir/`.

3.5 System Architecture

An overview of the whole system is shown in Figure 6. It consists of the PPIR system described in Section 3.4 connected to an OpenLDAP server by means of a *shell backend*. The OpenLDAP server has a variety of ways to access the actual data it provides LDAP access to. One of them is to run

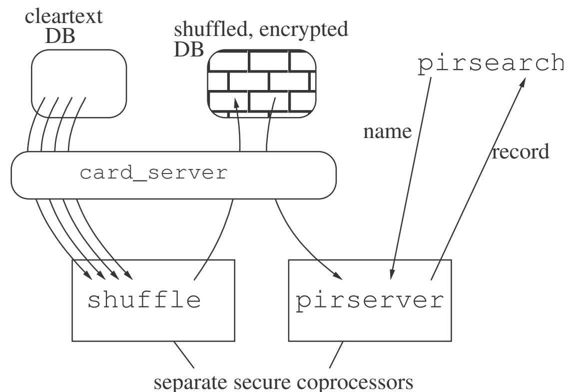

Figure 4: An overview of our PIR prototype. The card programs are `shuffle` which does the shuffling, and `pirserver` which handles retrievals. On the host, `pirsearch` performs a search by passing the name to `pirserver`, and `card_server` handles DB access requests from the card programs. Communication between the card and `card_server` is over SCC sockets, one of the mechanisms provided for 4758 Linux to talk to the outside. We serialize data using the External Data Representation (XDR) library in the RPC package.

a shell command to retrieve or update records.[10] We wrote a perl script to allow the OpenLDAP server to use our `pirsearch` program (see Figure 4). We tested this whole setup by sending LDAP queries from the Sylpheed[11] mail client to our PIR prototype.

This setup is a temporary way to achieve the connection to LDAP, and it is clearly not *root-secure*—secure even against an adversary running as root on the host—as queries are in the clear on the host before being handed to `pirsearch`. Our plan for a secured connection all the way from the client to the retrieval coprocessor is shown in Figure 7. It will make use of LDAP over SSL, and use OpenLDAP libraries for parsing of LDAP queries, and construction of LDAP responses.

4 Experimental Results

4.1 Performance

One of the main purposes of our prototype was to get a feel for how this PIR scheme performs in practice in the credential server setting, and what may need to be improved to make it really usable. The most interesting source of performance numbers was from the shuffling step, which is shown in Table 1. The database size N was 1000, and hashing had resulted in every bucket holding 5 records. The times shown are for one pass of the shuffle algorithm, where all the buckets are read by the card in order to keep one of them to write to a given position in the shuffled database.

The times for a run of 300 retrievals is shown in Figure 8.

[10] There are more operations besides query and update which are idiosyncratic to the LDAP protocol.

[11] `http://sylpheed.good-day.net/`

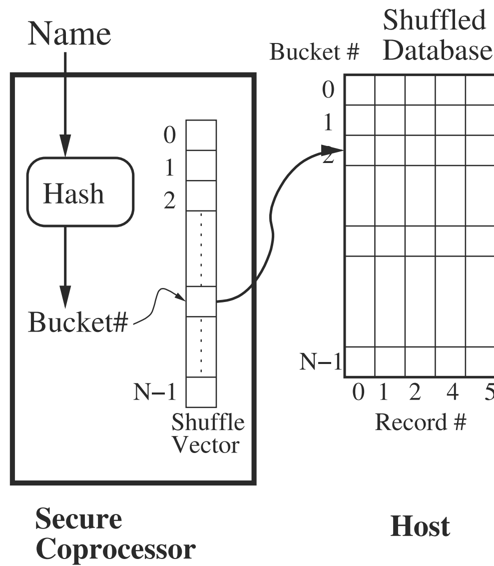

Figure 5: The Retrieval Procedure. Once the card has identified the correct bucket number, it retrieves all the records in that bucket, and keeps the correct one.

Record size (bytes)	**Time to Read 1000 Records (sec)**
115	18
530	24
1345	34

Table 1: Read time during shuffling vs. Record size.

4.2 Hashing

The main price of using a fixed hash function is that collisions inevitably occur, and in this case they are particularly damaging—since all buckets need to look the same to the host, they must all hold the same number of records. In our case the largest bucket received 5 records from the hash function, so we had to pad *all* the buckets to 5 records, thus having 4N dummy records—4000 for our test database.

5 Analysis

5.1 Prototype's Shuffling

Several observations arise from the figures in Table 1. Firstly, a linear relationship between the record size (s) and read time for 1000 records (t) is $t \approx 16 + \frac{s}{75}$. This confirms that there are considerable overhead costs to the host-card communication, and that maximizing the amounts of data transferred at a time is desirable.

Secondly, the whole shuffle, which consists of N scans through the whole database, would in this

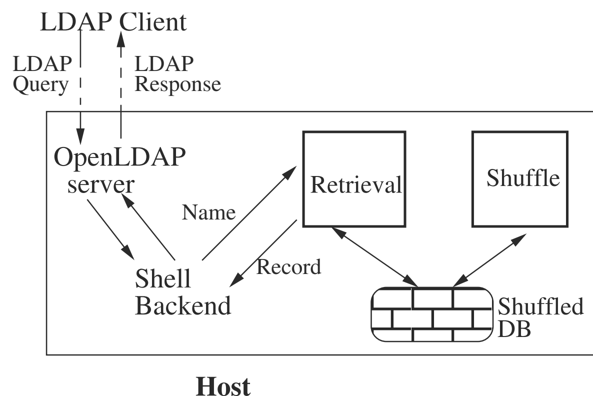

Figure 6: System Architecture. OpenLDAP is used to provide the gateway between our PPIR server and LDAP clients.

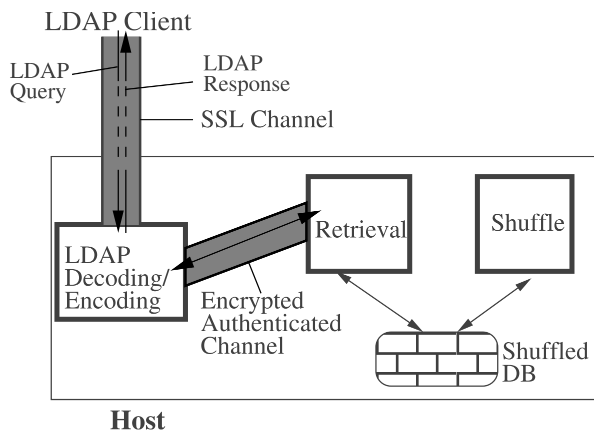

Figure 7: Final System Architecture, using a third coprocessor to handle LDAP and SSL operations. A separate coprocessor will likely be needed for these tasks because of space restrictions inside the coprocessors.

case take 18,000 seconds, or 5 hours, for the smallest record size. This brings into question the real usability of the scheme with naive shuffling for larger but quite realistic database sizes like 10,000 records—the prediction in that case is 500 hours, or almost 3 weeks—shuffle that!

5.2 Retrievals

From Figure 8, around the 300$^{\text{th}}$ retrieval the time reaches 3 seconds, which is a reasonable ceiling on the duration of a query, so we may want to set up the system so that a single shuffled database is used for about 300 queries.

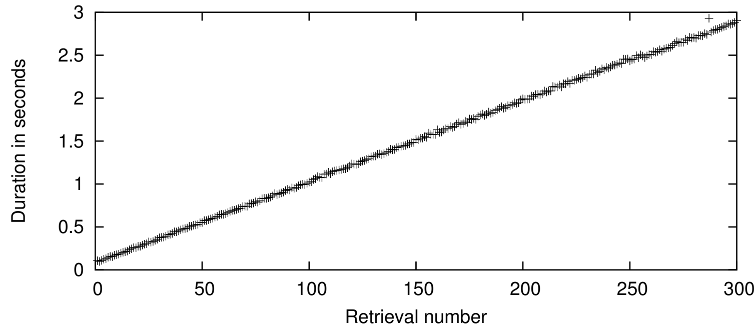

Figure 8: Times for each of 300 sequential retrievals of random records in the database.

5.3 Permutation Network Shuffling

A more detailed analysis of the expected running time of a permutation network is as follows. From the recursive construction of Figure 3, we can determine that a permuter with $N = 2^n$ inputs will have $2n-1$ columns of $N/2$ switches each. If the switches are executed in column major order, each column will consist of N record reads, and N record writes. Now if we approximate a running time for one column as twice the time for reading N records, this should amount to about 36 seconds for our N=1000 database. With $2\lceil \log N \rceil - 1 = 19$ columns, the switch execution stage should last about 12 minutes, and it should dominate the shuffle time, so 20 minutes is a conservative estimate of the total shuffling time.

For a larger database with say $N = 10,000$, each column of the permuter should take about 6 minutes, and there will be $14 \times 2 - 1 = 27$ columns, for a total of 160 minutes spent permuting, so perhaps 3 hours for the whole shuffle.

5.4 Name Resolution

As we wrote above, hashing required us to introduce 4N dummy records into the hashed database. This brings about a factor of 5 increase in the running time of most procedures in the system. If we use a perfect hash function, each record name would hash to a unique record number, and there would be no need to carry the deadweight of dummy records. Thus our current running times for shuffling *and* retrieval could be reduced by up to a factor of 5. The cost would be the complexity of computing the perfect hash function when the name set changes (which should be infrequent).

In Table 2 we list a summary of our measured and predicted shuffle run times.

5.5 Consolidation and Feasibility

If shuffling with a permutation network is combined with a name lookup method with less overhead than hashing with chaining, the shuffle time for a $10,000$ record database may be lowered from 3 to about one hour. In addition, reducing the hashing overhead should reduce retrieval times, as no dummy records would need to be fetched. The largest number of retrievals off one shuffled database should go up from 300 to perhaps 1000. If we have C coprocessors shuffling databases in parallel, they can produce C shuffled databases an hour, which give $1000C$ queries. The system would be able to deal with $1000C/3600 \approx C/4$ queries per second, answering each query in less than 3 seconds.

Scheme / DB Size	A	B	C
1,000	5 hrs	20 mins	6 mins
10,000	3 weeks	3 hrs	1 hr

Schemes:
A—Naive Shuffling, Hashing with chaining
B—Permutation Network Shuffling, Hashing with chaining
C—Permutation Network Shuffling, Low Overhead Name Resolution

Table 2: All Shuffling Times in One Place. The 5 hours figure was measured. 3 weeks is a prediction of our prototype's time on larger input. The other numbers are predictions of schemes we have analyzed in Section 5 and will be implementing.

Interesting to note here is that there is no point in having a big collection of shufflers, as the retrieval coprocessor will not be able to deal with much more than one query in 3 seconds on average—if the query rate goes higher when retrievals are taking close to 3 seconds, a queue will quickly build and the response time will be really bad. Thus, the shuffling will no longer be such a bottleneck, and parallelism (for sustaining a higher query rate) can be achieved by duplicating shufflers as well as retrievers.

6 Future Work and Conclusions

We currently have a functional prototype of a private credential directory accessible over LDAP. It has some fairly serious performance shortcomings, which we will be addressing in our next version. In particular we shall implement a faster database shuffling algorithm, and faster resolution of record names. We strongly believe, on the basis of our current measurements and the details of our proposed changes, that these changes will yield usable performance. We will connect this next version of our prototype to the certificate directory currently being rolled into operation for Dartmouth's new campus PKI, and so get real usage experience for the system. Dartmouth also plans to deploy a Shibboleth prototype, so we will have a testbed for a private Shibboleth AA. We believe that this system has realistic potential to address the problems of server privacy exposed at the beginning of the paper.

Acknowledgments The authors have received support from the Mellon Foundation, the NSF, AT&T/Internet2, and the U.S. Department of Justice (contract 2000-DT-CX-K001). The views and conclusions do not necessarily reflect those of the sponsors.

References

[1] Masayuki Abe and Fumitaka Hoshino. Remarks on mix-network based on permutation networks. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 317–334. Springer, 2001.

[2] Dmitri Asonov and Johann-Christoph Freytag. Almost optimal private information retrieval. In *Privacy Enhancing Technologies*, LNCS, San Francisco, 2002. Springer.

[3] Stefan Brands. *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy.* The MIT Press, August 2000.

[4] Stefan Brands. A technical overview of digital credentials. At `http://www.credentica.com/technology/technology.html`, Feb 2002.

[5] C. Cachlin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, LNCS. Springer-Verlag, 1999.

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

[7] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms*, chapter 27. McGraw-Hill, second edition, 2001. Problem 27-3 on permutation networks.

[8] Marlena Erdos and Scott Cantor. Shibboleth architecture. Available from `http://shibboleth.internet2.edu/`, May 2002. Version 5.

[9] Bob Jenkins. Minimal perfect hashing. `http://burtleburtle.net/bob/hash/perfect.html`, 2003.

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

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

[12] 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).

[13] 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.

[14] Eli Upfal. A permutation network. `http://www.cs.brown.edu/courses/cs253/slide/class2.ps`, 2000. Course Lecture Notes.

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

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