RC 21806 (Log#98098) (07/27/2000)
Computer Science/Mathematics

IBM Research Report

Practical Private Information Retrieval with Secure Coprocessors

S.W. Smith, D. Safford

IBM Research Division
T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

This page intentionally not blank.

Practical Private Information Retrieval with Secure Coprocessors

Sean W. Smith* Dave Safford†
IBM T.J. Watson Research Center

July 27, 2000

Abstract

What does it take to implement a server that provides access to records in a large database, in a way that ensures that the this access is completely private—even to the operator of this server? In this paper, we abstract this problem to a real world computer security application, and examine the question: using current commercially available technology, is it *practical* to build such a server, for real databases of realistic size, that offers reasonable performance? We consider this problem in the light of commercially available *secure coprocessors*—whose internal memory is still much, much smaller than the typical database size—and construct an algorithm that both provides asymptotically optimal performance, and also promises reasonable performance in real implementations.

1 Problem

1.1 Motivation

What does it take to implement a server that provides access to records in a large database, in a way that ensures the complete privacy of this access (and, potentially, the contents of the records themselves)—even to the operator of this server?

Access privacy alone would benefit many real-world scenarios:

- **Patent Information.** Data mining on a competitor's patent searches could shed useful light on their confidential research projects.
- **Maps.** Oil companies might rather their competitors not know their latest drilling locations.
- **Medical Records.** Unethical employers might wish to know how often a potential employee's medical records have been accessed—since frequent access might indicate a potentially expensive health problem.

Many other scenarios would benefit from *content privacy* as well as *access privacy*. For example:

- **Archives of Human Rights Abuses.** Suppose the server is seized (or the operator is served with a subpoena or a sufficiently large bribe) by an adversary interested in some particular subset of records.
  - The users who worked with those records would benefit if the adversary cannot link a record to them.
  - Furthemore, activists in a particular human rights case would benefit if the adversary can neither read any records relevant to that case, nor even learn if any such records exist in the system.

*Now on the faculty of the Department of Computer Science, Dartmouth College, 6211 Sudikoff Lab, Hanover NH 03755-3510 USA; sws@cs.dartmouth.edu.

†Global Security Analysis Lab, IBM T.J. Watson Research Center, Yorktown Heights NY 10598-0704 USA; safford@watson.ibm.com

- **"Privacy Act" Databases.** Root-secure access privacy and content privacy would benefit applications with large amounts of personally identifiable information, where the entity administering the application has strong motivation to suppress insider abuse.

  For example, consider a tax authority, where auditors (with special authorization) can examine the tax records of specific individuals.

  - Root-secure access privacy would ensure that even root on the server cannot know who is being audited.
  - Root-secure content privacy would ensure that even root on the server cannot reveal individual records without authorization.

- **"Vegetarian" Data Exchanges.** If a data exchange service ensures privacy of access and contents even from the server operator with full "root" privileges, then it can arguably also ensure privacy from any "Carnivore"-like automated analysis tool the operator may be compelled to install.

- **Private File Exchange Services.** A group wishing to set up a private file exchange service might prefer to honestly say they do not know which of their users has been accessing pirated MP3 files—or even if there are any MP3 files, pirated or not, in the service. (We discuss the legal and ethical implications—and some ideas for addressing them—in Section 5.1.)

Furthemore, we note that variations where users may *update* records needs content privacy—for otherwise, root would know which record the user touched, because it could see the changed plaintext.

In this paper, we abstract this problem to a real world computer security application, and examine the question: using current commercially available technology, is it *practical* to build such a server, for real databases of realistic size, that offers reasonable performance? We consider this problem in the light of *secure coprocessors* that have recently become available [4]—which provide a safe haven to execute code (and carry out high-speed symmetric cryptography). However, building a practical server using these devices creates a challenge: how to provide reasonable performance for databases typically much, much bigger than the internal memory of these devices.

This Paper The remainder of Section 1 discusses the context of this problem. Section 2 then introduces the more specific problem we focus on—private information retrieval using secure coprocessors—and then derives the theoretical optimal efficiency for this model. Section 3 presents two algorithms: a straightforward one that does not scale, and a more subtle one that achieves this optimal (theoretical) efficiency. Section 4 discusses the practical implications of implementing this algorithm, including performance estimates. Section 5 presents some avenues for future work.

1.2 Root Security

We abstract the privacy properties we desire for this service into a term we call *root-security*: an adversary (even with the equivalent of UNIX "root" privileges on the host) who cannot break the cryptography we use should not be able to learn what record i was requested in a particular query, nor even learn indirect statistics such as "πi is the most popular record requested" or "users who request πi usually also request πj" (for some permutation π, possibly unknown). For a service with the stronger property of content privacy as well as access privacy, the adversary should also not be able learn the plaintext contents of any particular record. (Clearly, if the adversary works with an authorized user, then she can learn what that user is authorized to learn).

With root-security, the query, the result, and all statistics should be secure even against traffic analysis and deliberate probing and memory manipulation on the host. However, we are not worrying about denial of service, nor about about hiding the fact that a query took place, nor—in this model—hiding who made the query.

We are interested in root-security for several reasons. First, it protects (maximally, by some metrics) the privacy of the users's actions: from the owners of the service, from hackers who may break into the service, from external parties who may compel the operator to provide inside access, and from adversaries who physically seize control of the machines. Additionally, by its maximal nature, root-security provides a level of privacy that may actually provide practical assurance—since history has shown that specifying weaker levels of security can open the door to unexpected statistical inference.

1.3 Previous Work

Previous research has explored related questions.

Private Information Retrieval (PIR) Previous theoretical work (e.g., [2]) has explored coding techniques by which a user can hide his queries from a distributed database. In this paper, we aren't are interested as much in the abstract problem but in its practicality: can we actually implement this with existing technology, and for realistic databases, and provide reasonable performance? This motivation provides us with goals that (for now) take us away from the focus of the earlier work. Such goals include:

- minimizing *user computation* (since no one wants to change their client too much);
- minimizing *user-server traffic* (since, for remote users, that's expensive);
- efficiently handling lots of queries at once;
- parallelizing well (so that throwing more hardware at it speeds things up); and
- using algorithms that depend on computation (such as streaming encryption) that our special-purpose technology can do quickly.

(However, we revisit these issues in Section 5.2.)

Oblivious RAM Previous theoretical work on *oblivious RAM* (e.g., [3]) addresses how to prevent instruction fetches from leaking execution details—but explicitly dismissed secure coprocessors as "infeasible."

Secure File Systems Previous work in *secure file systems* (e.g., [1]) and *cryptopaging* (e.g., [10]) protects database privacy against theft, but not against a malicious root. (Indeed, [8] inquired about how adversaries might learn internal operational details from observing cryptopaging details.)

Anonymizers Previous work in anonymizers (e.g., [7]) protects the privacy of *who* is taking some action. Root-security addresses the complementary problem of protecting *what* the action is.

2 Retrieval using Secure Coprocessors

2.1 Background: Secure Coprocessors

As noted earlier, the secret weapon we bring to this problem is a high-performance secure coprocessor: a general-purpose computer that can be trusted to carry out its computation unmolested, even if the adversary has direct physical access to the device.

Theoretical work on oblivious RAM [3] observed that selling "physically protected special-purpose computers for each task" would enable 'trivial" but "infeasible" solutions to problems in securing computation. However, subsequent secure coprocessor research has advanced the state of the art: hardware solutions are now feasible—but not quite trivial.

Smith and Weingart [9] showed how to build a generic secure coprocessor platform that third-party application developers could then transform into such special-purpose devices. This research then culminated in a family of commercially available devices [4], which feature—in a PCI form factor—a general-purpose computing environment (99Mhz 486-class CPU, megabytes of memory), physical and logical security protection validated at FIPS 140-1 Level 4—as well as hardware 3DES and SHA, and a FIFO structure to allow fast data movement through these elements [6].

2.2 Design Challenges

Because of these advances in secure coprocessing technology, it is now feasible (on a hardware level) for any researcher with a few $10K of funds to build a private information server by taking a host machine with ample PCI slots, and inserting these coprocessors.

The primary challenge is how to design the *software* that handles this task with reasonable performance, while preserving root security, when the database is typically too large to fit in any or all of the coprocessors.

However, the researcher would also need to address several additional challenges, including:

- what combination of cryptographic design and implementation optimizes coprocessor performance for this application;
- how such a service should authenticate its users;
- what access policy should govern who reads and writes this data;
- whether content privacy is important, or just access privacy suffices;
- how to efficiently implement key and database management information across the coprocessors;
- how to accomodate the possiblity that any one coprocessor might fail and lose all state;
- what legal and ethical issues arise (Section 5.1), and how to address them.

2.3 Retrieval in this Model

System Assumptions We abstract the specific problem of coprocessor-based information retrieval to the following model. (We consider the most general case: a server that provides both content privacy as well as access privacy.)

A single server has a number of secure coprocessors, and provides a query service for a number of records. Each record is stored as a whole unit on some suitable high-performance (but not necessarily secure) media, outside of the coprocessors. The stored records are encrypted and authenticated (see the discussion of $\mathcal{E}$ and $\mathcal{D}$ below). Figure 1 sketches this architecture.

We assume a secure coprocessor model based on the commercially available device: where the symmetric encryption engine can be configured in series with FIFOs, and thus the time complexity for encryption/decryption (and verification) can be modeled solely by the per-byte data transit rate.

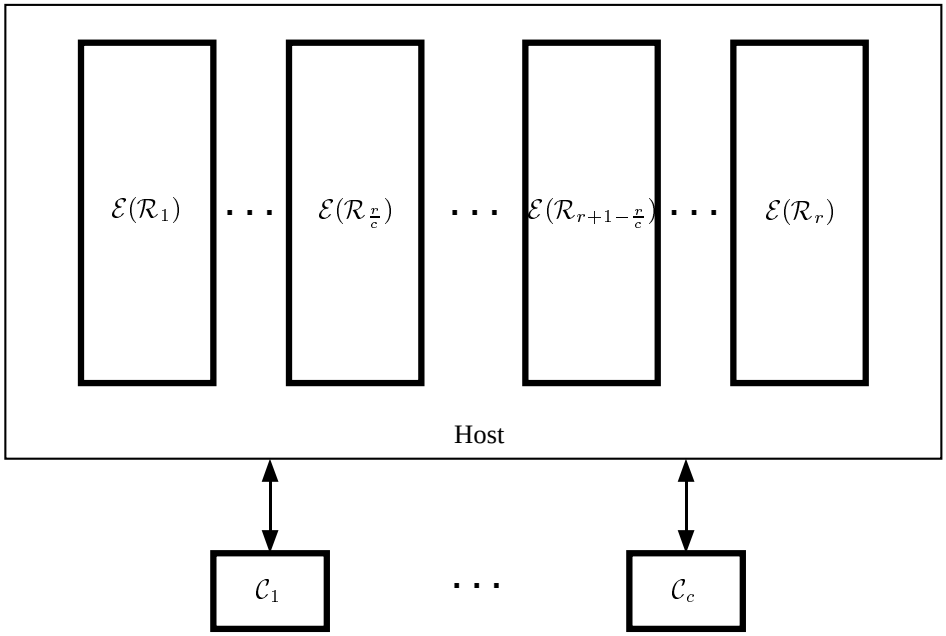

Figure 1 System architecture for coprocessor-based retrieval: a host stores r encrypted and authenticated records of size S_R each, but has c secure coprocessors (with internal memory S_C) to assist in private retrieval. However, $rS_R >> cS_C$: the database is much larger than the collective space of the coprocessors.

Parameters Formally, we describe the problem with the following parameters:

- The server has r records, $\mathcal{R}_1, ..., \mathcal{R}_r$.
- Each record is padded out to some maximum S_R bytes.
- The server has c coprocessors, $\mathcal{C}_1, ...\mathcal{C}_c$. (We assume $\mathcal{C}_1$ is designated as the *master* coprocessor for the server.)
- Internal coprocessor data memory has size S_C bytes. We assume that $rS_R >> cS_C$ (and it may very well be the case that even $S_R > S_C$).
- The server has received requests for q queries.

Cryptography Let $\mathcal{E}$ and $\mathcal{D}$ be authenticated encryption and decryption functions (respectively) based on a suitably secure symmetric cipher. For example:

- $\mathcal{E}$ might consist of appending a SHA-hash or CBC-DES-MAC, then encrypting the result using outer-CBC TDES. $\mathcal{D}$ consists of decrypting, then verifying the hash or MAC.
- $\mathcal{E}$ and $\mathcal{D}$ might instead consist of TDES using recent advances [5] in serial (or even parallelizable) chaining that provides authentication as well.

Throughout Section 3 and the remainder of Section 2, "encryption" and "decryption" refer to operations with $\mathcal{E}$ and $\mathcal{D}$, respectively. (If the service did not provide content privacy, then we would need to separate integrity checking from encryption, and—assuming that just doing an integrity check is cheaper—use just the former on the stored records.)

The Problem We can think of a query Q as a pair $(i, \mathcal{K})$ of record index and session key. A user communicates a query to the master coprocessor $\mathcal{C}_1$. After some degree of computation, the server returns to the user $\mathcal{E}_{\mathcal{K}}(\mathcal{R}_i)$: the desired record, encrypted under the specified session key $\mathcal{K}$. In the general case, the server with its coprocessors needs to be able to handle up to q queries simultaneously.

Parameters in Previous Work Previous work in private information retrieval usually characterizes the problem in terms of a database of n bytes, with k servers (who usually are assumed not to talk each other). In those terms, $n = rS_R$, but $k = 1$, since we only have one server. Furthermore, since we want this to be practical and practical users do not like to do extra work, we want to restrict the user's computation to the above two steps: establishing a session key and record number, and then receiving and decrypting the desired record.

2.4 Theoretical Lower Bound

In our initial analysis, we permit the system the luxury of accepting and processing the queries as a batch, but nevertheless follow the above storage model in which no information is cached inside the coprocessors across more than one batch.

In any root-secure algorithm for this model, each byte in each encrypted record must be read by at least one coprocessor when answering the set of q queries. (Otherwise, if part of some $\mathcal{R}_i$ was not read, then the adversary would know that $\mathcal{R}_i$ was not one of the requested records.) Thus, any algorithm meeting these conditions must process rS_R byte through the symmetric cipher.

Furthermore, each of the requested records must be re-encrypted for the requestors. This is an additional qS_R bytes.

Since the bytes can be processed across c coprocessors, and we are assuming that the coprocessor time complexity can be modeled by simple data transit rate, we have that any algorithm satisfying these conditions must have asymptotic time complexity bounded below by:

$$\Omega\left(\frac{(r+q)S_R}{c}\right)$$

3 Algorithms

To simplify exposition, we start with a straightforward but inefficient algorithm for coprocessor-based retrieval, (Section 3.1), and then move to the asymptotically optimal one (Section 3.2).

For simplicity, we also assume that the queries each request a different record. (Section 4.1 later will discuss how to handle the more complex case of multiple queries for the same record.)

3.1 Straightforward but Inefficient

We begin by considering the most straightforward algorithm: each record $\mathcal{R}_i$ is stored encrypted as a separate ciphertext $\mathcal{E}(\mathcal{R}_i)$ (computed using secret keys by the coprocessors but not by the host, obviously).

Figure 2 illustrates this algorithm.

An Easy Case To start with, let us consider an easy case:

- $q = 1$
- $c = 1$.
- $S_R \leq S_C$.

To handle query $Q_1 = (i_1, \mathcal{K}_1)$, one coprocessor can simply follow the following the algorithm:

- For $1 \leq i \leq r$, have each $\mathcal{E}(\mathcal{R}_i)$ streamed in through the symmetric engine.
- If $i = i_1$, then save these bytes in internal memory
- If $i \neq i_1$, then throw them away (but take the same out of time as it would to save them).
- When all r records have been processed, then $\mathcal{R}_i$ is in internal DRAM; stream back out through the symmetric engine, encrypting under $\mathcal{K}_1$.

This straightforward handling of this easy case takes time $O(rS_R)$: so far, so good.

More Coprocessors When $c > 1$, then each coprocessor can scan $\frac{1}{c}$ of the records. However, we then have a problem. Only *one* of these coprocessors has the right answer. But if, for some j, we don't read S_R bytes from coprocessor $\mathcal{C}_j$, then the adversary will know that the queried record is not in the jth $\frac{1}{c}$ of the records.

Consequently, we need to break the algorithm into two steps:

- the *streaming* phase, where each coprocessor reads in its share of the encrypted records, then outputs either the encrypted answer or encrypted nonsense;
- then the *combination* phase, where we must combine this partial result by selecting one of these c records in a root-secure way.

This straightforward approach to this harder case yields

$$O\left(\frac{rS_R}{c} + cS_R\right)$$

Bigger Records If $S_R > S_C$, then each coprocessor must now store its temporary state in an off-card cache. Each step of "stream in a record" must bring in both of these, and re-encrypt and output one—multiplying the time necessary for each such step by a factor of 3.

More Queries When we consider the fully general case (with $q > 1$), we run into complications.

During the initial streaming phrase, each coprocessor, in order to process its $\frac{r}{c}$ records for q queries, must either go through the records q times, or go through them once but process q cached copies at each step (or some combination thereof). During the streaming phase, each coprocessor thus ends up handling $O(\frac{qrS_R}{c})$ bytes somehow.

During the combination phase, we then need to select q of qc records. This appears to take at least qcS_R bytes.

Thus, the straightforward approach to the fully general case yields suboptimal complexity of

$$O(\frac{qrS_R}{c} + qcS_R)$$

3.2 Subtle, with Optimal Efficiency

We now present a more efficient algorithm, and start immediately with the general case: $q \geq 1$.

General Idea Upon analysis, the straightforward approach to the fully general case is slow for two reasons:

- In the streaming phase, because $S_R > S_C$, each coprocessor must handle qS_R bytes 3 times for each record.
- In the combination phase, because any one coprocessor could potentially have all q records, we need to look at all qS_R bytes from each coprocessor, in order to break any potential causality.

To overcome these problems, we developed an alternate way to subdivide the records so that:

- During the streaming phase, each record is small enough so that essentially $qS_R \leq S_C$, so each coprocessor need only handle $(q + r)S_R$ bytes.
- During the combination phase, no causality need to be broken—so at worst, this only requires re-encryption of the qS_R bytes to be returned to the users.

Striped Data The key to obtaining this efficiency to abandoning the idea of storing and processing data as whole records.

Let $S_S \leq \frac{S_C}{q}$, so q stripes fit inside one coprocessor. Instead of storing and processing the records as a sequence of whole records, we organize them as a sequence of *buckets* of *stripes*. The ith bucket consists of the ith stripe (that is, the ith S_S bytes) of each record. (See Figure 3.)

The Streaming Phase As with the easy case in Section 3.1 above, each coprocessor hanles a bucket by streaming it in one stripe at a time. If the stripe belongs to a record that is being queried, then the coprocessor saves it in internal memory; otherwise, the coprocessor discards it. (However, this operation is coded so that both options take the same time.) When the bucket is done, the coprocessor has q stripes in its memory; it re-encrypts each with the appropriate $\mathcal{K}_i$ and outputs them. Thus, the time per bucket is $(r + q)S_S$.

Since the total number of buckets is $\frac{S_R}{S_S}$ and each coprocessor handles $\frac{1}{c}$ of the buckets, this gives a net cost for the streaming phase of

$$\frac{S_R}{S_S} \cdot \frac{1}{c} \cdot (r + q)S_S \quad = \quad \frac{(r + q)S_R}{c}$$

Notice that not only is this the asymptotic optimum, but we also have a constant of one: if we model coprocessor performance by data transfer rate through the engine, then we need only multiple the above by the engine's time-per-byte to get a time estimate.

The Combination Phase However, a significant advantage of striping is that the combination phase becomes very simple, since *there is no potential causality to break*. In the earlier whole-record algorithm, if the adversary can observe whether or not a query result came from the output of a particular coprocessor after the streaming phase, then the adversary can learn whether nor that requested record was in that coprocessors $\frac{1}{c}$ of the records. But in this striping algorithm now, the adversary already knows that, after a given coprocessor processes a given bucket $\mathcal{B}_i$, this coprocessor will spit out the ith stripe of each requested record—and this does not help, since the bucket the coprocessor examined contained the ith stripe of *every* record.

In the striping algorithm, at the end of the streaming phase, coprocessor $\mathcal{C}_j$ has output the jth $\frac{1}{c}$ of each record (as a sequence of separately encrypted stripes). Combination can consist of merely concatenating them at no additional cost (if the users are satisfied with receiving ciphertext with a new IV each stripe), or of re-reading and re-encrypting the responses, at a cost of another $O(\frac{qS_R}{c})$ bytes. Either leaves the asymptotic cost of the striping algorithm at the theoretical optimum.

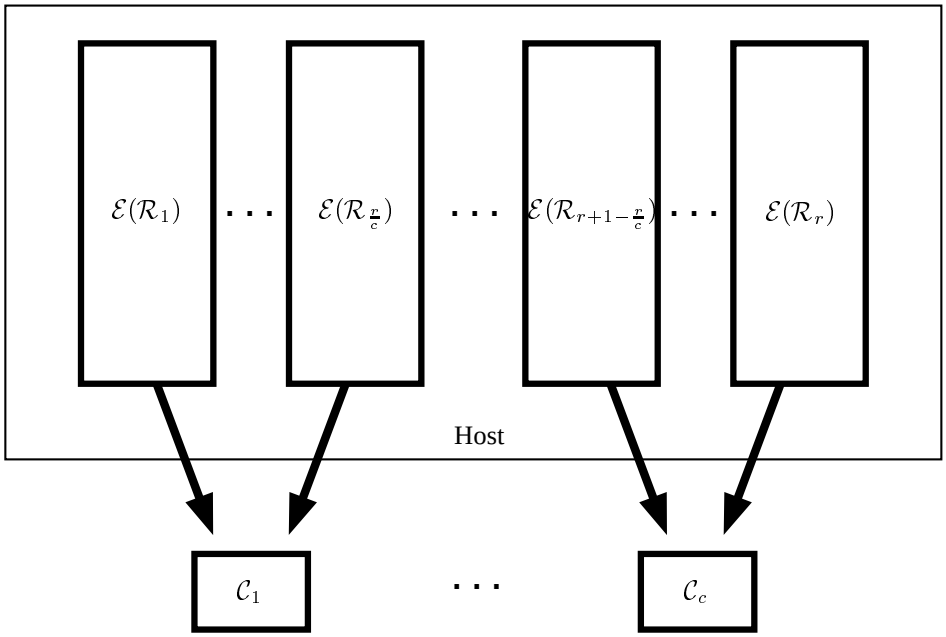

Figure 2 In the straightforward algorithm, each coprocessor looks at $\frac{1}{c}$ of the records. However, this does not scale well, since each coprocessor outputs q records, and we still need to select q of these qc.

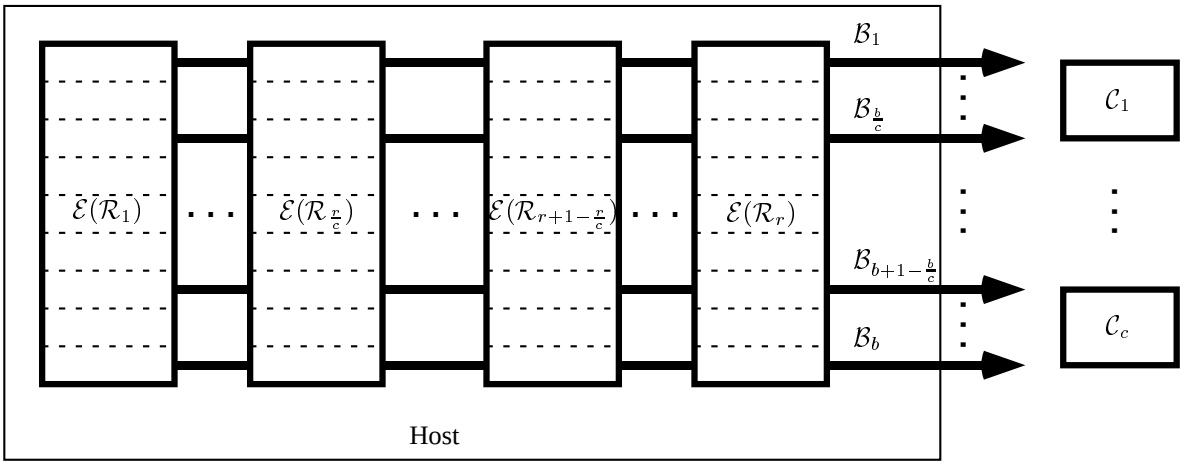

Figure 3 We obtain much better efficiency and scaling by dividing each record into b stripes, and having each coprocessor look at a bucket $\mathcal{B}_i$ of stripes—one from each record—at a time. With this approach to streaming, the combination phase becomes trivial—since no causality needs to be broken.

4 Implementation Details

4.1 Coding Issues

Error Detection and Active Attacks So far, we have focused primarily on using symmetric cryptography, for secrecy of records against a passive adversary. The realities of accommodating storage/transmission errors—and an *active* adversary who might deliberately tamper with data—required that we also consider using redundancy of some type to detect and suppress such errors—as Section 2.3 noted.

One issue we did not consider was who should respond to an authentication error, and how; the answers are relevant to preserving privacy.

- If each coprocessor detects and responds to errors on a *bucket* granularity (independent of whether or not the error was in an interesting stripe), then an active adversary can learn nothing, even in a coalition with users.
- If the user then detects an error, he can request retransmission of the post-coprocessor output without revealing which record he was interested in—because if the adversary had introduced an error on the way *into* the coprocessor, the coprocessor would have detected it.

How to structure this redundancy and how to check are not issues for asymptotic complexity, but are for practical performance—for example, we would rather not have a 3X blow-up in data handling by having coprocessors read a stripe into a buffer, then crank it through DES to check a DES-MAC, then crank it through again to get a new DES-MAC for the user. The commercially available device [4] includes hardware support for 3DES and SHA-1; we plan to explore the potential of streaming data through both engines simultaneously.

Redundant Queries At first glance, the possibility that two or more of the q queries may request the same record complicates the equi-time "save-or-discard" operation at the heart of the streaming step—because (to preserve root security) each operation would need to take $\Omega(q)$ time. However, many straightforward techniques eliminate this: for example, in the striping algorithm, the coprocessor saves each record of interest, and then (when outputting the stripes from the bucket) re-encrypts a common record for each separate query.

Dynamic Bucket Size In Section 3.2, we chose S_S and hence bucket size based on q. However, in a real system, the number of queries active at any one time is likely to vary. We can accommodate this by choosing some maximum number of queries q', and organizing the data in stripes based on this count. For any given q then, we simply be sure to read in a whole number of q'-buckets in each bucket step.

Asynchronous Queries The model of Section 2.3 implicitly assumed that all q queries show up at the same time. In reality, they may show up at different times. Since, essentially, the coprocessors are just cycling through the data and there is no natural reason why any particular record is denoted as $\mathcal{R}_1$, straightforward extensions of this algorithm should allow dynamically adding queries during execution.

Reducing Storage We (and others) normalize all record lengths to a maximum S_R not just because it makes analysis easier, but because otherwise the adversary could deduce query information based on size of encrypted record. But this normalization leads to much wasted space and time, since short records much be padded out. While it appears inevitable that the encrypted response to any given query must be S_R bits, we could reduce a lot of storage and processing time in the striping algorithm—if we don't mind giving away some information about the distribution of record sizes in the database—by not padding the stored records. (That is, coprocessors might read in shorter buckets, and still output qS_S bytes.) If S_U is total size of the unpadded records, then the time complexity would go down to

$$\frac{S_U + qS_R}{c}$$

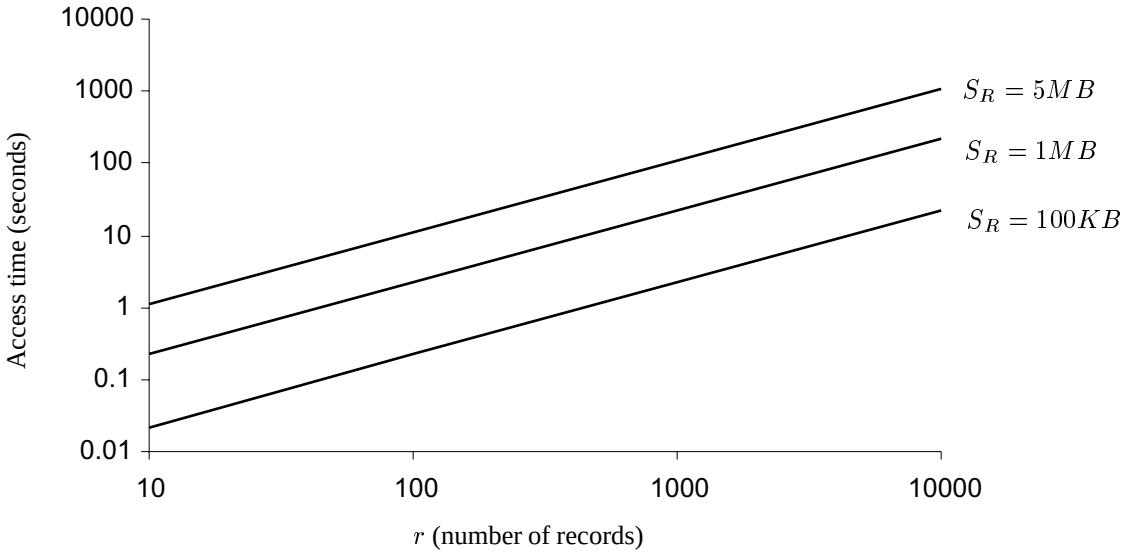

Figure 4 Wild estimate of performance, for 5 coprocessors and at most 10% of records being simultaneously requested.

Private Information Storage So far, we have dealt exclusively with retrieving records. The algorithm ought to extend easily to inserting and modifying records as well (although that would raise questions of freshness).

4.2 Performance Estimates

Now we come to the punchline. We started this investigation because we wanted to implement this, and wondered: was this at all practical?

Let's consider the commercially available device, the IBM 4758 Model 2. If we were putting together a server just for this purpose, we might be able to arrange 5 free PCI slots, for $c = 5$. The standard commercial software for the 4758 turns it into a generic cryptographic accelerator; previous experimental work [6] in modifying software for alternate cryptographic usages suggests that a streaming data rate (for large stripes) in the 10 megabyte/second range might be feasible. (Of course, this previous work also showed that bottlenecks were never where one thought they would be.) Let's also assume that $q < \frac{r}{10}$. Figure 4 summarizes the estimated performance here, for three different record sizes (including 5 megabytes, the approximate size of an MP3 compression of a typical rock music CD).

For an extremely large case, we estimate it would take under 31 hours to return a 5 gigabyte movie, from a collection of 1000 movies. It is interesting to note that this processing time is quicker than the typical download time. Interleaving the bucket processing (so that coprocessor $\mathcal{C}_i$ handled buckets $\mathcal{B}_i, \mathcal{B}_{i+c}, \mathcal{B}_{i+2c}$, etc.) would enable the server to transmit stripes as soon as they came out of the streaming phase.

It is also interesting to note that, due to the way that striping enables parallelization, we can also improve throughput by using more host machines. E.g., using ten hosts with five coprocessors each would (by this estimate) improve all these times by a factor of ten. (In this case, the encrypted buckets themselves would be divided among the ten hosts.)

Again, the next step in establishing the validity of these projections is to actually implement the data structures and a variety of encryption and authentication schemes, and then measure. These experiments will also quantify the performance cost (if any) that content-privacy adds.

5 Future Work

The obvious next step is to implement and measure (even via simulation—although we plan to build a prototype using real coprocessors).

However, many additional avenues of future work suggest themselves.

5.1 Ethical and Legal Implications

Building and deploying a root-secure database service raises some potential ethical issues. For a timely example, it would enable someone to set up a service that allows users to download MP3 compressions of recorded songs, while making it impossible for recording artists to determine which of these downloads violated copyright laws.

One might characterize solutions to such problems as *selective weakening* of root security. For example, the community in the above scenario might decide that an acceptable arrangement is that the service provider pay royalties for the frequency of access to copyrighted songs, and in turn prohibit users from downloading more than some maximum number of these in any given one-week period.

Our use of secure coprocessors to provide full root-security provides an interesting avenue to implement such selective weakenings: since we already have trusted third parties (the coprocessors) with full plaintext access, we can implement such policy solutions as computation alone, instead of via more complex cryptographic schemes that change with each new policy.

In a similar vein, the use of secure coprocessors also provides promising ways to address other problems that appear much more awkward in non-coprocessor cryptographic techniques. For example:

- providing flexible key recovery schemes;
- preserving privacy of user actions while providing atomicity against various failures;
- balancing privacy with marketing services—e.g., the *coprocessor* could track a user's purchases and offer him or her special deals based on these patterns, but this information would be hidden from root.

5.2 Experimental Evaluation of Previous Theory

In this paper, we presented an algorithm that is linear in the total size of the database, but which meets our practicality goals (and is "linear" with a small constant, in the computation that these devices are quick at). Prior work in single-server PIR might fit our framework, with the coprocessor functioning as a proxy for the PIR's user and the host functioning as the PIR server. Furthemore, our striped-bucket approach should extend to add parallelization to these more complex schemes (e.g., we replace the streaming phase with an instance of PIR on smaller records).

However, it's not clear how quickly the special purpose devices could carry out this work, or whether things would scale well to more queries, and parallelize well with more coprocessors. These are all interesting areas for exploration and experiment.

5.3 Violating Assumptions

Our theoretical lower bound of Section 2.4 depended on two assumptions of our model: that each record was stored as a whole unit, and that these unuits were stored outside the coprocessors. If we violate these assumptions—either by allowing caching or (as in the PIR work) storing data in more complex, inter-dependent ways—can we get better performance?

5.4 Anonymization

As noted in Section 1.3, this paper addresses a problem that is complementary to the problem of hiding user identities. It would be interesting to combine our work with a CROWDS-like anonymity scheme, to provide privacy for the entire interaction.

6 Conclusion

From this analysis, we conclude that practical private information indeed appears feasible with commercially available secure coprocessor technology.

In some sense, what we are doing is extending the limits of secure coprocessing. Secure coprocessors provide—if the physical security assumptions hold—a haven where *details* of internal computation are hidden even from a dedicated adversary. In this paper, we have explored (for a sample problem) how to preserve this property while extending the file system to the host and the computation across several coprocessors. One wonders at the implications of more general "secure multi-processing."

Acknowledgments

The authors gratefully acknowledge helpful discussions with Nao Itoi, Peter Gutmann, Mark Lindemann, Charles Palmer, and Michael Waidner.

References

[1] R. Anderson, R. Needham, A. Shamir. "The Steganographic File System." D. Ausmith, ed..: *Information Hiding: Second International Workshop IH98*. Portland, Oregon, USA.

[2] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan. "Private Information Retreival." *Journal of the ACM*. 45: 965-982. November 1998.

[3] O. Goldreich, R. Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." *Journal of the ACM*. 43: 431-473. May 1996.

[4] *IBM4758 Models 2 and 23 PCI Cryptographic Coprocessor.* IBM Product Brochure G221-9091-01.

[5] C. S. Jutla. *Encryption Modes with Almost Free Message Integrity.* Draft Research Report, IBM T.J. Watson Research Center, July 2000.

[6] M. Lindemann, S.W. Smith. *Improving DES Hardware Throughput for Short Operations.* Research Report RC-21798, IBM T.J. Watson Research Center. July 2000.

[7] M. Reiter and A. Rubin. *CROWDS: Anonymity for Web Transactions.* DIMACS Technical Report, August 1997.

[8] S. W. Smith. *Secure Coprocessing Applications and Research Issues.* Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory. August 1996.

[9] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks (Special Issue on Computer Network Security.)* 31: 831-860. April 1999.

[10] B. S. Yee. *Using Secure Coprocessors.* Ph.D. thesis. Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University. May 1994.