Practical server privacy with secure coprocessors

by S. W. Smith
D. Safford

What does it take to implement a server that provides access to records in a large database, in a way that ensures that this access is completely private—even to the operator of this server? In this paper, we 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—scaling well, parallelizing well, working with the current client infrastructure, and enabling server operators of otherwise unknown credibility to prove their service has these privacy properties? 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. Preliminary prototypes support this analysis, but leave many areas for further work.

This paper presents some of our work in using *commercial off-the-shelf* (COTS) secure coprocessors to enhance privacy and security of servers in general, and our consideration of private information retrieval in particular. We start by setting the broad context of this research effort. Is there a practical way to systematically add privacy to real distributed information services?

This question has many aspects. To begin with, the World Wide Web is currently the pre-eminent medium for distributed information services. If we want to design a practical system, it had better fit within this medium by using the existing client infrastructure; by minimizing changes to current server infrastructure; by maintaining reasonable server performance, at realistic workloads; and by being deployable with currently existing, commonly available technology.[1]

However, discussions of Web security and privacy usually focus on just a few areas: authentication of the server, encryption of the client-server traffic, and potential server use of cookies. These discussions overlook a more fundamental issue: participants in distributed Web services are forced to trust the integrity of the server. That is, participants must trust that the server works as advertised, that it keeps private client data private, and that it otherwise behaves correctly. Given that the current Web public key infrastructure (PKI) establishes little more than server identity, and that the Web creates a global marketplace where clients may have no additional information about a server operator, these issues are critical. Stakeholders include the clients whose interests directly depend on these privacy and security properties of the server. Stakeholders also include the server operators themselves, who may gain a competitive or legal advantage by being able to establish, with high assurance, that their service can be trusted—even though they may have motivation to subvert it.

As an extreme end point, we say that a server is *root-secure* with respect to certain properties when an ad-

versary—even with the equivalent of UNIX** "root" privileges on the host—who cannot break some level of cryptography is not able to subvert them. We are interested in root security for several reasons. First, it protects—maximally, by some metrics—the privacy of the user'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, because history has shown that specifying weaker levels of security can open the door to unexpected compromise.

In broad strokes, then, much of our current research[1,2] focuses on developing practical techniques—that is, with minimal change to the current paradigm—to add provable root security to servers even when run by operators of otherwise unknown credibility.

The enabling technology. The secret weapon we bring to this family of problems 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.

Yee's seminal work[3] demonstrated the potential of such devices. Smith and Weingart[4] showed how to build a generic secure coprocessor platform that third-party application developers could then transform into such special-purpose devices. This research culminated in the family of commercially available devices, the IBM Cryptographic Coprocessor.[5] These devices feature—in a PCI (Peripheral Component Interconnect) form factor—a general-purpose computing environment (99 MHz 486-class CPU, megabytes of memory), physical and logical security protection validated at FIPS (Federal Information Processing Standard) 140-1 Level 4[6], as well as hardware 3DES (Triple Data Encryption Standard) and SHA (Secure Hash Algorithm), and a FIFO (first-in-first-out) structure to allow fast data movement through these elements.[7]

The coprocessor hardware architecture provides a general-purpose computing environment for applications, with hardware support for cryptographic applications. However, the device also provides crucial security features. Continuously active tamper-detection circuitry monitors tamper detectors and, in case of physical attack, destroys sensitive secrets in secure memory before an adversary can access them. Hardware locks protect crucial code and secrets from possibly malicious or faulty application code, preserving the ability of each device to properly authenticate its configuration, and preventing a device with a rogue application from impersonating other devices and applications.

The coprocessor[5] features a software architecture that permits application developers to install and update their applications onto these devices at customer sites, in a way that protects the privacy and security interests of the developers, the customers, and IBM. The Model 2 family of the IBM 4758 includes full support for outbound authentication, which enables onboard applications to authenticate, to remote entities, their identity and their status as applications running on untampered hardware.

The software and hardware architecture that support outbound authentication take into account the possibility that malicious code may run at root level and that a corrupt version of the code-loading code may be released. See our paper[4] on the architecture for more details. The architecture extends to handle the trust issues introduced by maintenance of lower-level code.

The retrieval problem. For this paper, we consider the specific problem of private information retrieval (PIR). What does it take to implement a server that provides access to records in a large database, in a way that ensures *access privacy* and, potentially, the privacy of contents of the records themselves, even to the operator of this server?

We apply the private information retrieval 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?

In a root-secure retrieval scheme, the adversary should neither be able to learn what record i was requested in a particular query, nor 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 to learn the plaintext contents of any particular record. However, content

privacy does have a limit: if the adversary works with an authorized user, then he or she can learn what that user is authorized to learn.

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

Motivation. Access privacy alone would benefit many real-world scenarios. In the domain of patent information, data mining on a competitor's patent searches could shed useful light on the competitor's confidential research projects. In the domain of mapping data, oil companies prefer that their competitors not know their latest drilling locations. In the domain of medical records, unethical employers might wish to know how often a job applicant's medical records have been accessed, because frequent access might indicate a potentially expensive health problem.

Many other scenarios would benefit from content privacy as well as access privacy. For one example, consider archives of human rights abuses and suppose the server is seized, or the operator is served with a subpoena, or is offered a sufficiently large bribe, by an adversary interested in some particular subset of records. The users who accessed those records would benefit if this adversary could not identify them. Furthermore, activists in a particular human rights case would benefit if the adversary could neither read any records relevant to that case, nor learn if any such records exist in the system.

In another domain, consider "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 the operator with root authority cannot know who is being audited. Root-secure content privacy would ensure that even the operator with root authority on the server could not reveal individual records without authorization.

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 wiretapping/analysis devices, such as the FBI's Carnivore tool (e.g., Reference 8), that the operator may be compelled to install.

Similarly, a group wishing to set up a private file exchange service might prefer not to know which users have been accessing pirated MP3 files—or even if there are any MP3 files, pirated or not, in the service. (In a later section, we discuss the legal and ethical implications and some ideas for addressing them.)

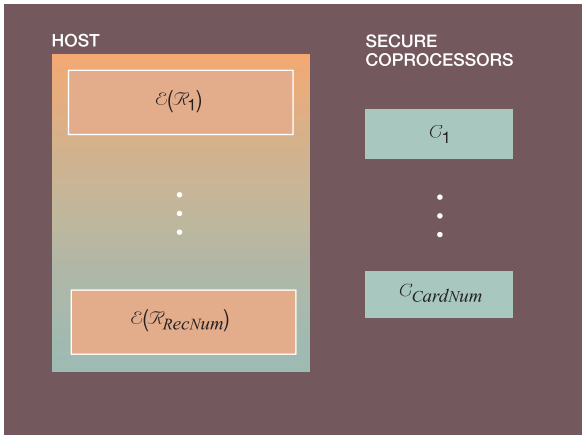
We note that systems where users may update records would require that records be stored in ciphertext, and that all records be re-encrypted after each update. Otherwise, the operator with root authority can learn about user action from observing which records change.

Previous research. Previous theoretical work in private information retrieval (e.g., References 9, 10) has explored coding techniques by which a user can query a distributed database but hide details of these queries from the database itself. In this paper, we are not interested as much in the abstract problem as 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 the client too much; minimizing user-server traffic, since, for remote users, that is expensive; efficiently handling many queries at once; parallelizing well, so that adding more hardware speeds things up; and using algorithms that depend on computation (such as streaming encryption) that our special-purpose technology can do quickly. We later revisit these issues.

Previous theoretical work on oblivious RAM[11] (random access memory) addresses how to prevent instruction fetches from leaking execution details, but it explicitly dismissed secure coprocessors as "infeasible." Previous work in secure file systems[12] and cryptopaging[3] protects database privacy against theft, but not against a malicious root. Indeed, other work[13] inquired about how adversaries might learn internal operational details from observing cryptopaging details.

Figure 1 System architecture for coprocessor-based retrieval



Previous work in anonymizers[14] protects the privacy of *who* is taking some action. Root security addresses the complementary problem of protecting *what* the action is. The implementation and use of mobile agents as a model for distributed information services is another area of complementary research[15,16] that suggests several areas for future work, such as exploring our server privacy techniques in order to protect agent privacy, and extending our techniques to provide privacy when the server handles requests not via a local database query, but via the dispatch of an agent.

This paper. We are interested in providing a root-secure retrieval service, using current secure coprocessor technology. In the next section, "Problem," we present this version of the retrieval problem and derive the theoretical optimal efficiency for this model. In the section "Algorithms" that follows, we present two algorithms: a straightforward one that neither scales nor parallelizes well, and a more subtle one that does and also achieves this theoretical optimal efficiency. Then, in the section, "System," we present the current status of our efforts to use these ideas to build a prototype of a complete privacy server system. We then conclude with avenues for future work.

Problem

As noted, secure coprocessors provide a safe haven in which to execute code and carry out high-speed cryptography. Their commercial availability changes the playing field. Previous work[11] dismissed using "physically protected special-purpose computers for each task" as "infeasible" but "trivial." The hardware approach is no longer infeasible: subsequent secure coprocessor research has advanced the state of the art, and now permits any researcher with a few US $10K of funds to build a private information server by taking a host machine with ample PCI slots and inserting these coprocessors.

However, an efficient realization of the hardware approach is elusive. 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. Obviously, we want to be able to handle any one query in a reasonable time. However, we also want performance to scale with the number of queries: the processing of Q queries should not multiply the query processing time by Q. Furthermore, we want to be able to exploit the parallelism offered by multiple devices: doubling the hardware should reduce the total time by two. This paper addresses these challenges.

The model. We abstract from the specific problem of coprocessor-based information retrieval to the model described below. We consider the most general case: a server that hides both access and content from its operator.

Our model consists of a single server that has a number of secure coprocessors, and that provides a query service to a database containing records. Each record is stored as a unit on some suitable high-performance, but not necessarily secure, storage medium separate from the coprocessors. The stored records are encrypted and authenticated. Figure 1 shows 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 internal/external data transit mechanisms and thus the time complexity for encryption/decryption can be modeled solely by the per-byte data transit rate.

Formally, we describe the problem with the following parameters. The server has *RecNum* records, $\mathcal{R}_1, \ldots, \mathcal{R}_{RecNum}$. Each record is padded out to some maximum *RecSize* bytes. The server has *CardNum* coprocessors, $\mathcal{C}_1, \ldots \mathcal{C}_{CardNum}$. We assume $\mathcal{C}_1$ is designated as the *master* coprocessor for the server. Each

internal coprocessor has a data memory of size *CardSize* bytes. We assume that

$$RecNum \cdot RecSize \gg CardNum \cdot CardSize$$

and it may even be the case that even $RecSize > CardSize$. We assume the server is processing *QueryNum* queries.

Let $\mathscr{E}$ and $\mathscr{D}$ be authenticated encryption and decryption functions, respectively, based on a suitably secure symmetric cipher. For example, $\mathscr{E}$ might consist of appending a keyed message authentication code (MAC), such as SHA-HMAC (hash-based MAC), then encrypting the result using Triple DES (3DES) in outer-CBC (cipher block chaining) mode. (We refer the reader to a standard cryptographic reference, such as Reference 17, for more discussion of these topics.) $\mathscr{D}$ consists of decrypting, then verifying the hash or MAC. Using a keyed MAC instead of a hash function frees us from potential attacks if the encryption function is not known to be nonmalleable. Alternatively, $\mathscr{E}$ and $\mathscr{D}$ might instead consist of 3DES using recently discovered chaining modes[18] that provide authentication as well. The fact that such chaining can be carried out on different portions of the data in parallel may also prove useful for our situation.

Throughout this paper, "encryption" and "decryption" will refer to operations with $\mathscr{E}$ and $\mathscr{D}$, 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.

For the actual problem, we can think of a query Q as a pair $(i, \mathscr{K})$ where i is the record index and $\mathscr{K}$ is the session key. A user-generated query is sent to the master coprocessor $\mathscr{C}_1$. After some processing the server returns $\mathscr{E}_{\mathscr{K}}(\mathscr{R}_i)$ to the user, that is, the desired record $\mathscr{R}_i$ encrypted under the specified session key $\mathscr{K}$. In the general case, the server and its coprocessors need to be able to handle up to *QueryNum* queries simultaneously.

Previous work in private information retrieval usually characterizes the problem in terms of a database of n bytes, with k noncooperating servers. For our model, $n = RecNum \cdot RecSize$, but $k = 1$, because we only have one server. Furthermore, we want this to be practical; hence, in contrast to previous research, we 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.

A theoretical lower bound. We now derive an asymptotic lower bound for the time complexity of providing private information retrieval in our model. That is, we present a function of the model parameters, such that, when all the parameters become sufficiently large, the actual time complexity is bounded below by this function multiplied by some constant. We refer the reader to a standard complexity reference, such as Section 2.1 in Reference 19, for more discussion of these topics.

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 each record is stored separately, and 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 *QueryNum* queries. Otherwise, if part of some $\mathscr{R}_i$ was not read, then the adversary would know that $\mathscr{R}_i$ was not one of the requested records. Thus, any algorithm meeting these conditions must process $RecNum \cdot RecSize$ byte through the symmetric cipher.

Furthermore, each of the requested records must be re-encrypted for the requestors. This is an additional $QueryNum \cdot RecSize$ bytes.

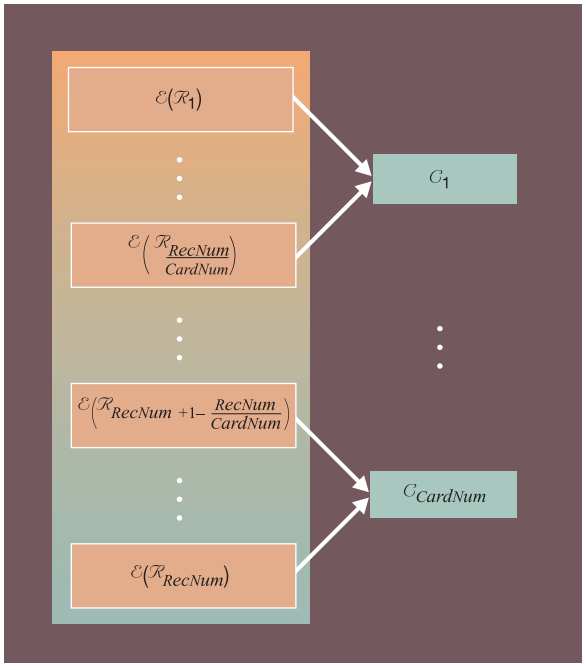
Since the bytes can be processed across *CardNum* coprocessors, and we are assuming that the coprocessor time complexity can be modeled by simple data transit rate, we have that any algorithm satisfying these conditions must have asymptotic time complexity bounded below by:

$$\Omega\left(\frac{(RecNum + QueryNum) \cdot RecSize}{CardNum}\right)$$

Algorithms

To simplify exposition, we start with a straightforward but inefficient algorithm for coprocessor-based retrieval, and then we describe an asymptotically optimal one.

Figure 2 Coprocessor-based retrieval using Algorithm 1



Algorithm 1. 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. The coprocessor just streams in the entire database, looking for the correct record.

Problems arise, however, when one tries either to handle more queries, or divide the work among multiple devices. Let us first consider an easy case. Suppose $QueryNum = 1$, $CardNum = 1$, and $RecSize \leq CardSize/2$. To handle query $Q_1 = (i_1, \mathcal{K}_1)$, one coprocessor can simply use the following algorithm:

- For $1 \leq i \leq RecNum$, 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 amount of time as it would to save them. The $CardSize/2$ assumption on record size means that, even if the architecture does not support anything more clever, we can always just bring the uninteresting records into a dummy buffer.
- When all *RecNum* records have been processed, then the record of interest $\mathcal{R}_{i_1}$ is in internal storage. We stream it back out through the symmetric engine, encrypting under $\mathcal{K}_1$.

Since our system model assumes encryption/decryption hardware in series with the internal/external data transit mechanism, this straightforward case requires transferring $RecNum \cdot RecSize$ bytes in, then sending *RecSize* back out. Thus, this straightforward handling of this easy case takes time $O(RecNum \cdot RecSize)$. So far, so good.

Let us now proceed to a more general case. When $CardNum > 1$, then each coprocessor can scan $1/CardNum$ of the records. However, we then have a problem. Only *one* of these coprocessors has the right answer. But if, for some j, we do not read *RecSize* bytes from coprocessor $\mathcal{C}_j$, then the adversary will know that the queried record is not in the jth $1/CardNum$ of the records. Figure 2 illustrates this approach.

Consequently, we need to break the algorithm into two phases: the *streaming* phase, where each coprocessor reads in its share of the encrypted records, then outputs either the encrypted answer or encrypted nonsense, where these are encrypted with some intermediate set of keys; and then the *combination* phase, where we must combine these partial results by selecting one of these *CardNum* records in a root-secure way. This straightforward approach to this harder case yields complexity

$$O\left(\frac{RecNum \cdot RecSize}{CardNum} + CardNum \cdot RecSize\right)$$

Let us now consider another generalization: bigger records. During the streaming phase, a coprocessor needs to consider two records: the current candidate and a placeholder for the correct record.

When *CardSize* was big enough, the coprocessor could store the placeholder internally and still have room to bring in each candidate one at a time. As a consequence, each step of "stream in a record" required one transfer of *RecSize* bytes, through the decryption engine and into the card. However, if $RecSize > CardSize$, then the placeholder must be stored externally, instead of in the card. Furthermore, the coprocessor had better read and rewrite this placeholder at each step, otherwise, it will reveal the identity of the record of interest. Hence, in addition to the transfer of *RecSize* bytes to bring in the record of interest, we must bring in the *RecSize* bytes of the

current placeholder, and then send *RecSize* bytes back out when we rewrite it. Thus, the constant on *RecSize* goes up to three. Again, recall that in our system model, the encryption/decryption engine can be configured in series with the transfer mechanism.

Finally, when we consider the fully general case (with $QueryNum > 1$), we run into even more complications. During the initial streaming phase, each coprocessor, in order to process its *RecNum*/*CardNum* records for *QueryNum* queries, must either go through the records *QueryNum* times, or go through them once but process *QueryNum* cached copies at each step (or some combination thereof). During the streaming phase, each coprocessor thus ends up handling $O(QueryNum \cdot RecNum \cdot RecSize/CardNum)$ bytes somehow. During the combination phase, we then need to select *QueryNum* of *QueryNum* · *CardNum* records. This appears to take at least *QueryNum* · *CardNum* · *RecSize* bytes.

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

$$O\left(\frac{QueryNum\ RecNum\ RecSize}{CardNum} + QueryNum \cdot CardNum \cdot RecSize\right)$$

Algorithm 2. We now present a more efficient algorithm and start immediately with the general case:

$QueryNum \geq 1$

Upon analysis, the above straightforward approach to the fully general case is slow for two reasons. First, in the streaming phase, because $RecSize > CardSize$, each coprocessor must handle *QueryNum* · *RecSize* bytes three times for each record. Second, in the combination phase, because any one coprocessor could potentially have all *QueryNum* records, we need to look at all *QueryNum* · *RecSize* bytes from each coprocessor. Not looking at all the output from a given coprocessor would let the adversary conclude that at least some of the records of interest did not come from that coprocessor's share.

To overcome these problems, we develop an alternate way to subdivide the records. Suppose that, during the streaming phase, each record is small enough so that essentially $QueryNum \cdot RecSize \leq CardSize$, and thus each coprocessor need only handle $(QueryNum + RecNum) \cdot RecSize$ bytes. Then during the combination phase, no correlation needs to be broken—so at worst, this only requires re-encryption of the *QueryNum* · *RecSize* bytes to be returned to the users.

The key to obtaining this efficiency is to abandon the idea of storing and processing a record's data as an atomic unit. Instead, we divide each record into *stripes* of size *StripeSize* bytes, as illustrated in Figure 3. Let $StripeSize \leq CardSize/QueryNum$, so *QueryNum* stripes fit inside one coprocessor. We organize the data processed by a coprocessor as a sequence of *buckets* of *stripes*, where the *i*th bucket consists of the *i*th stripe of each record and the number of buckets is

$B = RecSize/StripeSize$

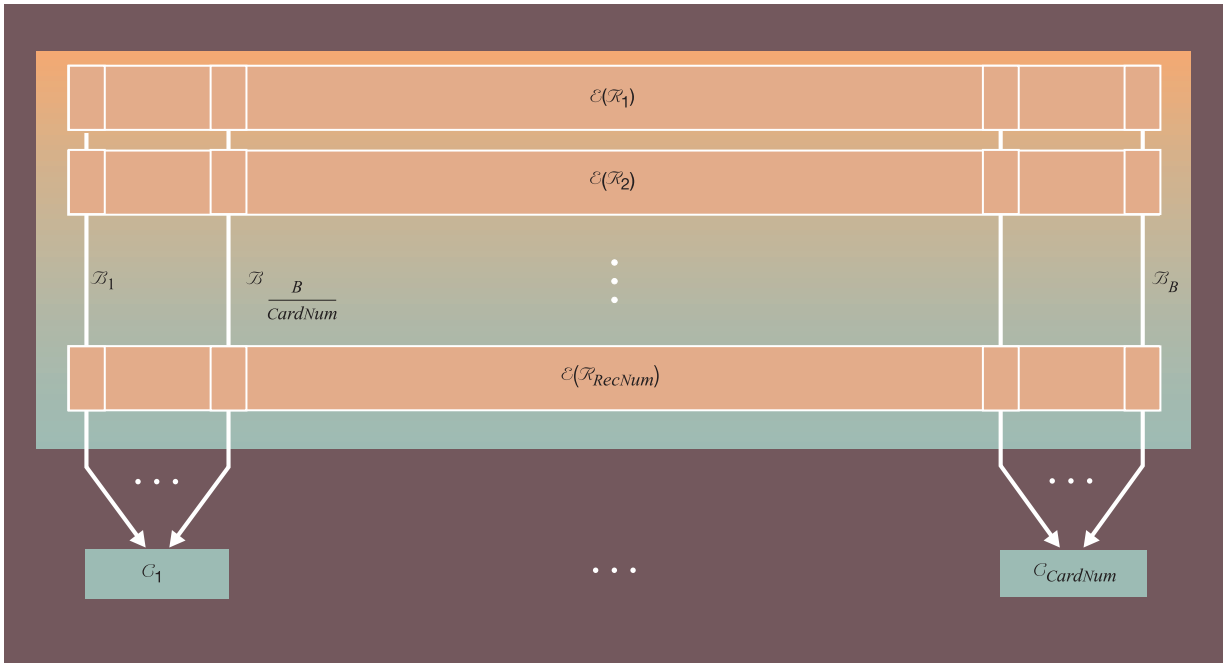
In the streaming phase of this algorithm, as with the straightforward algorithm presented earlier, each coprocessor handles a bucket by streaming it in one stripe at a time. If the stripe belongs to a record that is being queried, then the coprocessor saves it in internal memory; otherwise, the coprocessor discards it. However, this operation must be coded so that both options take the same time; otherwise, an adversary could observe the difference. When the bucket is done, the coprocessor has *QueryNum* stripes in its memory; it re-encrypts each with an appropriate key and outputs it. Thus, the time per bucket is $(RecNum + QueryNum) \cdot StripeSize$. Since the total number of buckets is *RecSize*/*StripeSize* and each coprocessor handles 1/*CardNum* of the buckets, this gives a net cost for the streaming phase of

$$\frac{RecSize}{StripeSize} \cdot \frac{1}{CardNum} \cdot (RecNum + QueryNum) \cdot StripeSize = \frac{(RecNum + QueryNum) \cdot RecSize}{CardNum}$$

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

However, a significant advantage of striping is that the combination phase becomes very simple, because

Figure 3 Coprocessor-based retrieval using Algorithm 2



we have no information to suppress. In Algorithm 1, 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 or not that requested record was in that coprocessor's 1/*CardNum* of the records. But in Algorithm 2, the adversary already knows that, after a given coprocessor processes a given bucket $\mathcal{B}_i$, this coprocessor will output the *i*th stripe of each requested record—and this does not help, since the bucket the coprocessor examined contained the *i*th stripe of every record.

Thus, in the striping algorithm, at the end of the streaming phase, coprocessor $\mathcal{C}_j$ outputs the *j*th 1/*CardNum* of each record, as a sequence of separately encrypted stripes. For the combination phase we consider two cases. If the users are satisfied with receiving ciphertext with a new initialization vector for each stripe, then combination consists of merely concatenating the stripes at no additional cost. Alternatively, combination consists of rereading and re-encrypting the responses, at a cost of another $O(QueryNum \cdot RecSize/CardNum)$ bytes. Note that in the first case, if consecutive stripes are concurrently encrypted by different coprocessors, then the last block of a stripe will not be ready in time to use as the initialization vector for the next. In general, the user would need to deal with *CardNum* independent chains. Either approach leaves the asymptotic cost of the striping algorithm at the theoretical optimum:

$$O\left(\frac{(RecNum + QueryNum) \cdot RecSize}{CardNum}\right)$$

System

Our motivation in exploring the server privacy issues is to come up with a practical solution to these privacy problems, using current COTS technology. Our ultimate goal is a complete working system, with reasonable performance, and a complete blueprint to enable anyone to repeat this work. In this section we present the current status of our prototype.

Basic service prototype. We have built a basic prototype that implements the striping approach. As is typical of coprocessor applications, we partition the code into the *card size* running inside the coprocessor, and the *host side* running on the host. In our pro-

totype, the host-side code runs on Linux platforms and handles an arbitrary number of coprocessors by launching a separate thread to handle each one. The card-side code runs in the application layer of standard production IBM 4758 Models 2 and 23 devices. This means the card-side code confines itself to the standard CP/Q++ embedded operating system provided with the IBM 4758. As noted below, we believe significant performance improvement will be possible with modifications to kernel-level code in the card. As part of this prototype, we built a framework that allows, via compile-time options, the host-side code to either work directly with real cards, or simulate a number of cards. In our initial prototype, we used outer-CBC 3DES as the symmetric cipher, and SHA-1 on the plaintext for redundancy.

To date, we have validated this prototype running it with one real card and with larger numbers of simulated cards. More work is in progress, as noted below.

Basic approach. In an idealized implementation, the card-side CPU just brings in stripes and saves or discards them. In our initial prototype, we had to work with the generic CP/Q++, which only lets the card-side application do data transfer and cryptography through a fairly limited API (application programming interface), which did not support this ideal vision. Furthermore, as previous work[7] shows, the CP/Q++ system has a per-request cost that penalizes multiple small requests.

Given these constraints, our prototype card-side code allocates an internal DRAM (Dynamic Random Access Memory) buffer that can accommodate an entire bucket, brings a bucket in at once (through 3DES), checks the SHA on each stripe, then saves, and internally re-encrypts, the interesting stripes in an internal output buffer. Doing a hash on each stripe saves us the trouble of recomputing it when we do the re-encryption. When the buffer is full, it blasts the contents back out to the host.

Our prototype also supports *asynchronous requests*. The system model we introduced earlier implicitly assumed that all *QueryNum* 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 $\mathcal{R}_1$, any given query can start at any point in the cycle.

Error detection and active attacks. Our algorithms section 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.

In our prototype, the host-side code runs on Linux platforms and handles an arbitrary number of coprocessors.

We needed to consider 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 or she can request retransmission of the postcoprocessor output without revealing which record he or she is interested in. This is so 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 for practical performance.

Performance issues. As long as the number of simultaneous queries are a small fraction of the number of records, our model suggests that the turnaround time for each query is approximately the time it takes to send each byte in the entire database into some card in the card farm, for symmetric cryptography and integrity checking.

The good news is that our prototype confirmed this. The bad news is that our bytes-per-second-per-card figure was in the 600–800 kilobytes/second range—disappointing for a device whose 3DES engine can exceed 18 megabytes/second! From our previous work[7] with performance optimization for this device, we speculate that the main bottleneck here is DMA (direct memory access) between the 3DES engine and the card's internal RAM. It is interesting to note that in our prototype, bringing in the data via DES, 3DES,

Table 1 Query turnaround time for current prototype and its future enhancement

	Current Prototype	Projection
Web pages	<2 seconds	<1 second
MP3 songs	26 minutes	<2 minutes

or no encryption at all, made no difference in the transfer time, supporting the hypothesis.

For another approach, the card's 3DES engine supports doing 3DES and SHA in series. If we could exploit this fact, we could decrypt and calculate a hash of the plaintext as part of bringing the data in—instead of decrypting on the way in, then hashing as a second step. This approach would bring the transfer speed up to 1.2 megabytes/second/card. The advertised API does not officially support this operation, but we may be able to exploit an unofficial mechanism by having our application bypass the official library and send a message directly to the CP/Q++ module in charge of the 3DES/SHA engine.

To summarize, our prototype established that the scheme works and that the limiting factor for performance is the transfer/encryption rate of the native hardware. Since the hardware is rated over 25 times faster than what we measured, however, we are optimistic that we can remove this bottleneck. In some sense, our work is hindered by the fact that the COTS hardware was not designed for the "external-to-internal-to-external" processing that this application requires.

Next steps. The most significant next step is to improve this per-card performance. Modifying kernel-level code in the card to allow the CPU to directly pull bytes from the 3DES engine via PIO (programmed input/output), and to do SHA in serial should significantly improve performance. Our earlier optimization work yielded a factor of 1000 improvement, but from a worse starting point; here, we anticipate a factor of 10. This kernel-level approach would also free us from having to allocate an internal DRAM buffer for a bucket, allowing us to accommodate larger buckets and fewer flushes of the output buffer.

We also want to run the prototype with larger numbers of real cards. We note that the PCI bus is rated in excess of 130 megabytes/second, and therefore, in theory, it should not be a bottleneck for any reasonable number of cards in a host. Furthermore, many new servers have multiple sets of PCI buses.

Projected performance. To make things concrete, let us assume we have a five-card set-up, and consider two data sets: 1000 5-kilobyte Web pages, and 1000 5-megabyte MP3 files. Let us also assume that $QueryNum < RecNum/10$. We then estimate the turnaround time for what the current prototype would support and what the system would support if we could improve the per-card performance to 10 megabytes/second—still barely half of its rated performance, as shown in Table 1.

Clearly, a critical barrier to the practicality of this solution, particularly for large record sizes, is improving the per-card performance!

If future technical experiments validate these estimates, we would then have to face the question: Are users and other stakeholders willing to accept these turnaround times in exchange for this level of privacy? We note that, in the projected case, we could service 1000 5-gigabyte movies in about 31 hours, which is less than the typical download time in the typical home dial-up connection.

The rest of the system. Our initial prototype focused on demonstrating that the design and the data structures worked. Our next steps will focus on improving the card performance. However, to show that a complete working system is possible, we also need to demonstrate that a real host can feed its cards quickly enough.

To offer this service in the real world, remote clients with legacy browsers need a root-secure way to interact with the system. In our related WebALPS project[1,2] we are modifying legacy Web servers to permit remote clients to establish certain SSL (secure sockets layer) sessions directly with an entity inside a coprocessor. WebALPS provides the necessary root-secure interface, and also enables many other exciting avenues to enhanced security and privacy of Web interactions.

The key element in making our design practical is enabling a service provider to prove root-secure privacy. Our security architecture and outbound authentication support for the IBM 4758 enables an on-card application from an officially sanctioned developer to request generation of key pairs certified to belong to that application, in that configuration, in that untampered card, and then to access pri-

vate-key services for these key pairs. This technology will enable the WebALPS guardian at the server site to prove that it is a bona fide front end to a bona fide privacy server, and also enable authentication and key management between the various cards in the server farm.

Future work and concluding remarks

This paper presents a snapshot of one aspect of research that is ongoing. The primary avenue of future work is to address the rest of performance, feature, and system issues discussed in the previous section. As noted previously, the major barrier to reasonable performance of this scheme for large record sizes is the limitations imposed by the production-level COTS devices: the components support sufficiently fast transfer and encryption, but the configuration and firmware (as shipped) do not. In the short term, we plan to try rewriting the kernel-level drivers inside the coprocessor. But in the long term, these issues would vanish entirely with a new hardware design around the existing encryption engines.

Database structure. The coprocessors need to access host-side data. However, once records are arranged in encrypted stripe sets, the host-side transfer overhead should be no different from the straightforward scheme. However, in this work, we have not considered how to structure the database to minimize the restructuring cost when coprocessors are added or removed.

Reducing storage. It is the practice to assume all records have length *RecSize*, not just because it makes analysis easier, but because otherwise the adversary could deduce query information based on the size of the encrypted record. But this leads to much wasted space and time, because short records must be padded out. While it appears inevitable that the encrypted response to any given query must be *RecSize* bytes, we could reduce a lot of storage and processing time in the striping algorithm, if we do not 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 *QueryNum* · *StripeSize* bytes. If *UnpaddedSize* is net size of all the unpadded records, then the time complexity would go down to

$$\frac{UnpaddedSize + QueryNum \cdot RecSize}{CardNum}$$

That is, *RecNum* · *RecSize* goes down to *UnpaddedSize*.

Private information storage. So far, we have dealt exclusively with retrieving records. The algorithm ought to extend easily to updating records as well. However, each "examine a bucket" step would also need to re-encrypt the bucket, to keep the host from learning which stripes were changed. This continual re-encryption suggests nontrivial freshness, key management, and host storage issues, and the extra transfers would reduce performance.

Although we have dealt so far with retrieving records from the database, the algorithm can be extended to updating of records.

Anonymization. As noted earlier, 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[14] to provide privacy for the entire interaction.

Additional server computation. The coprocessor approach provides a trusted computational entity with full knowledge of user activity. This fact may provide promising ways to address other problems that are more difficult to handle in designs without cryptographic coprocessors. Such problems include: providing flexible key recovery schemes, preserving privacy of user actions while providing atomicity against various failures, and balancing privacy with marketing services. As an example solution for the last problem, 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.

Experimental evaluation of 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 computation at which these devices are quick. However, there exist a number of theoretical results for various other settings of this PIR problem. It would be very interesting to explore implementing these in this

client-server-coprocessor framework. For example, prior work in single-server PIR might directly fit our framework, with the coprocessor functioning as a proxy for the PIR user and the host functioning as the PIR server. Furthermore, our striped-bucket approach should extend to add parallelization to these more complex schemes. For example, we could replace the streaming phase with an instance of PIR on smaller records. However, it is 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.

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: because 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.

Broader research issues. While privacy of retrieval is of interest, we are also interested in the broader issue of how to improve security and privacy of distributed information services, in practical ways, with minimal deviation from the current infrastructure. As part of the newly established Dartmouth College PKI Lab, we are currently exploring a number of areas, using coprocessors (e.g., the WebALPS project) and other techniques.

Concluding remarks. We have shown that practical server privacy 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 multiprocessing."

Acknowledgments

The authors gratefully acknowledge helpful discussions with Nao Itoi, Peter Gutmann, Mark Lindemann, Charles Palmer, Ron Perez, and Michael Waidner, and helpful comments from the referees.

This work was supported in part by Award No. 2000-DT-CX-K001 from the National Institute of Justice, Office of Justice Programs.

**Trademark or registered trademark of The Open Group.

Cited references

1. S. W. Smith, *WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions*, Research Report RC-21851, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (October 2000).
2. S. Jiang, *WebALPS Implementation and Performance Analysis*, Technical Report TR2001-399, Department of Computer Science, Dartmouth College, Hanover, NH (June 2001).
3. B. S. Yee, *Using Secure Coprocessors*, Ph.D. thesis, Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University, Pittsburgh, PA (May 1994).
4. S. W. Smith and S. H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor," *Computer Networks* (Special Issue on Computer Network Security) **31**, 831–860 (April 1999).
5. *IBM4758 Models 2 and 23 PCI Cryptographic Coprocessor*, G221-9091-02, IBM Corporation (2000).
6. S. W. Smith, R. Perez, S. H. Weingart, and V. Austel, "Validating a High-Performance, Programmable Secure Coprocessor," *22nd National Information Systems Security Conference*, National Institute of Standards and Technology, Washington, DC (October 1999).
7. M. Lindemann and S. W. Smith, "Improving DES Hardware Throughput for Short Operations," *USENIX Security Symposium*, August 2001, to appear (a preliminary version is available as IBM Research Report RC-21798).
8. J. Schwartz, "Computer Security Experts Question Internet Wiretaps," *The New York Times*, December 5, 2000.
9. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private Information Retrieval," *Journal of the ACM* **45**, 965–982 (November 1998).
10. C. Cachin, S. Micali, and M. Stadler, "Computationally Private Information Retrieval with Polylogarithmic Communication," EUROCRYPT 1999, Springer-Verlag, Berlin (1999).

11. O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM* **43**, 431–473 (May 1996).
12. R. Anderson, R. Needham, and A. Shamir, "The Steganographic File System," D. Aucsmith, Editor, *Information Hiding: Second International Workshop IH98*, Portland, Oregon, Springer-Verlag, Berlin (1998).
13. S. W. Smith, *Secure Coprocessing Applications and Research Issues*, Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, Los Alamos, NM (August 1996).
14. M. Reiter and A. Rubin, *CROWDS: Anonymity for Web Transactions*, DIMACS Technical Report, Center for Discrete Mathematics & Theoretical Computer Science, Rutgers, NJ (August 1997).
15. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik, "Itinerant Agents for Mobile Computing," *IEEE Personal Communication Systems* **2**, 34–49 (October 1995).
16. B. S. Yee, *A Sanctuary for Mobile Agents*, Computer Science Technical Report CS97-537, University of California, San Diego, CA (April 1997).
17. A. Menezes, P. Oorschcot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL (1997).
18. C. S. Jutla, "Encryption Modes with Almost Free Message Integrity," *Cryptology ePrint Archive*, Report 2000/039 (2000).
19. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA (1990).

Accepted for publication April 17, 2001.

Sean W. Smith *Department of Computer Science, Dartmouth College, 6211 Sudikoff Laboratory, Hanover, New Hampshire 03755 (electronic mail: sws@cs.dartmouth.edu).* Dr. Smith is interested in the practical and theoretical aspects of security and reliability in distributed computation. As a postdoctoral fellow and staff member at Los Alamos National Laboratory, he performed security reviews and designs for a wide variety of public sector clients. As research staff member at the IBM Thomas J. Watson Research Center, he designed the security architecture for (and helped code, test, and validate) the IBM 4758 secure coprocessor. Since July 2000, Dr. Smith has been on leave of absence from IBM, in order to teach and do research at Dartmouth College. Dr. Smith was educated at Princeton and Carnegie Mellon Universities, and is a member of ACM, USENIX, Phi Beta Kappa, and Sigma Xi.

David Safford *IBM Research Division, Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: safford@watson.ibm.com).* Dr. Safford is manager of the Global Security Analysis Laboratory, which performs research in security of networked systems, including vulnerability analysis, security auditing and intrusion detection tools, cryptographic coprocessors, and secure operating systems. His current research interest is adding strong security features, such as mandatory access control and mandatory authentication, to open source operating systems. He received his Ph.D. degree from Texas A&M University in 1990.