Toward Practical Private Access to Data Centers via Parallel ORAM[*]

Jacob R. Lorch, Bryan Parno, James Mickens *Microsoft Research*
Mariana Raykova[†] *IBM Research*
Joshua Schiffman *AMD*

Abstract

Recent events have shown online service providers the perils of possessing private information about users. Encrypting data mitigates but does not eliminate this threat: the pattern of data accesses still reveals information. Thus, we present Shroud, a general storage system that hides data access patterns from the servers running it, protecting user privacy. Shroud functions as a virtual disk with a new privacy guarantee: the user can look up a block without revealing the block's address. Such a virtual disk can be used for many purposes, including map lookup, microblog search, and social networking.

Shroud aggressively targets hiding accesses among hundreds of terabytes of data. We achieve our goals by adapting oblivious RAM algorithms to enable large-scale parallelization. Specifically, we show, via new techniques such as *oblivious aggregation*, how to securely use many inexpensive secure coprocessors acting in parallel to improve request latency. Our evaluation combines large-scale emulation with an implementation on secure coprocessors and suggests that these adaptations bring private data access closer to practicality.

1 Introduction

Using systems like GFS [2] and Haystack [3], companies store petabytes of sensitive user data, including emails, photos, and social networking activity. Early cloud stores optimized for the traditional metrics for distributed file systems: performance, availability, and scalability. However, *information privacy* has become increasingly important. Mutually distrusting users expect the cloud provider to prevent unauthorized cross-user data access. However, users may also distrust the cloud provider itself. For example, unscrupulous employees of the provider may try to inspect user data for criminal purposes [4]. Even if the provider is honest, its treasure trove of private data is alluring to hackers, who can leverage stolen data for political or financial gain [5].

Applications like the Persona social network [6] operate solely on encrypted user data, preventing the service provider from directly inspecting sensitive information. However, even if data is encrypted, user *access patterns* can leak important information [7–9]. For example, a malicious observer can use access patterns to correlate two users' activities, e.g., to discover friendships in a social network or to tell whether users are geographically close. Even visiting the service via an anonymous proxy [10] does not obviate this threat, since the attacker may have auxiliary information. For example, if he knows where the victim lives, he can observe when map tiles in that location are fetched, then correlate subsequent accesses to learn the victim's work address. Even without data access, an observer can see how often each item is accessed, then use statistical inference to deduce the contents of encrypted items. For instance, one study identified over 80% of encrypted email queries based on access pattern alone [7].

We thus introduce Shroud, a cloud-based storage substrate that hides user access patterns from the servers that store user data. Shroud implements a distributed block interface [11] that allows clients to read and write to addresses 1 to N, where N is large, e.g., 2^{35}. Shroud's block interface is oblivious—storage servers cannot learn: the plaintext of user data, the addresses requested, nor relationships between requested addresses. Shroud's block interface is directly usable by services, like maps, that require simple, one-to-one bindings between names and objects, e.g., from geographic coordinates to map tiles. Services with richer semantics can layer complex data structures atop Shroud, as ext3 and NTFS do atop a physical disk's block interface.

To hide user access patterns, Shroud leverages oblivious RAM (ORAM) algorithms [12–20]. Prior ORAM algorithms cannot satisfy the performance and scalability demands of cloud applications: these algorithms handle thousands instead of trillions of items, and they offer access latencies of minutes or hours. In contrast, we set aggressive goals: on modern hardware, Shroud targets low-latency access times for storage of tens of billions of 10KB blocks. For this, Shroud uses several techniques.

First, Shroud uses many secure coprocessors acting in parallel as client proxies in the data center. Clients establish secure connections with the proxies, which return the small results of expensive message exchanges between the proxies and cloud servers.

Second, using a novel intra-data-center communication protocol, *oblivious aggregation*, Shroud implements a distributed, parallel ORAM algorithm that remains secure even when the adversary controls the network that connects the proxies and servers.

Finally, Shroud includes novel techniques for defending against malicious servers (§5.3) and coprocessor failures (§5.4), both practical realities that are often ignored or glossed over in the theoretical literature.

[*] This is the full version of our conference paper [1].

[†] Supported by NSF Grant No. 1017660

Unlike nearly all other ORAM work, we implement our algorithms on real, secure hardware and run at large scale. Implementation reveals critical bottlenecks not apparent in previous big-O analysis or simulations, such as the I/O bottleneck of modern secure hardware (§4.2).

In summary, Shroud is the first practical system for oblivious data access at data-center scales.

2 Background and Related Work

We briefly survey prior work on cloud file systems, ORAM, and private information retrieval (PIR). For details on PIR and ORAM, we defer to surveys [20, 21].

2.1 Cloud File Systems

There are a variety of enterprise-scale systems for storing and analyzing large amounts of data [2, 3, 22–26]. These systems emphasize massive scalability, high availability, and robustness to multiple server-side failures, and they are generally designed to scale using cheap commodity hardware.

Cloud storage systems export a variety of interfaces to clients. For example, FDS [23] is a simple, fast blob store that requires clients to implement richer semantics like locking schemes. S3 [25] defines a key-value interface. HDFS [24], Cosmos [22], and GPFS [26] export a hierarchical namespace, but only the latter is fully POSIX-compliant. Shroud's simple block-level API is closer in spirit to low-level interfaces like those of FDS and S3. However, prior work like Petal [11] demonstrates that many useful services can be built atop a distributed block abstraction. Note that none of the systems above provides oblivious data access.

2.2 Private Information Retrieval (PIR)

Traditional PIR. In a private information retrieval (PIR) protocol [27], a user retrieves block i from a database of N items held by a server, without the server learning i. Sion et al. [28] argue that in practical settings, existing PIR schemes will never be more efficient than the trivial PIR scheme of downloading the entire database. However, they evaluate number-theoretic solutions; techniques based on other assumptions, e.g., lattice-based linear algebra schemes [29], can outperform the trivial PIR scheme [30]. Nonetheless, Olumofin and Goldberg's results [30] indicate that even these schemes require over 25 minutes to access one block from a 28GB database over a home network connection, suggesting single-server PIR is not yet ready for the massive (terabyte or petabyte) data sets and sub-second requirements of the data center (§4.1).

Multi-Server PIR. Using multiple non-colluding servers can improve the efficiency of both information theoretic and computational solutions [21, 27, 30, 31]. Nonetheless, these response times are still too slow for data-center workloads, and more importantly, the data sets we consider (§4.1) are large enough to make duplication prohibitive and are held by companies with little incentive to allow others to duplicate their data.

Trusted Hardware PIR. A more practical approach to PIR is for the user to trust a secure coprocessor installed at the server. The user sends an encrypted request to the coprocessor, which uses an Oblivious RAM (ORAM) protocol to access the requested block and return it to the user via the secure channel (Fig. 1). This means that the user employs essentially the same amount of bandwidth as it would for a non-private request. It is also more efficient than performing the ORAM operations between the user and the server. Finally, it allows many independent users to use the same service.

Iliev and Smith first proposed this elegant combination of trusted hardware and ORAM for PIR [32], though the notion is implicit in the earlier work of Smith and Safford [33] and Asonov and Freytag [34].

2.3 Oblivious RAM (ORAM)

Classic. Goldreich and Ostrovsky [12–14] introduced ORAM as a mechanism by which a trusted processor could make use of an untrusted RAM. Most existing ORAM solutions use the basic memory structure suggested by Ostrovsky's "Hierarchical Scheme" [13, 14]. The ORAM is arranged in a series of progressively larger caches. Each cache consists of a hash table of buckets. When a block is requested, the algorithm checks a bucket at each level of the hierarchy. If the block is found, the search continues, to hide the location where the block was found, but looks for a dummy block instead. Finally, the block is reinserted into the top-level cache. When a cache is close to overflowing, it is obliviously shuffled into the cache below it. This scheme and the modern schemes derived from it require serial operation for security, making it challenging to leverage a large number of distributed secure coprocessors.

Modern. Recent ORAM work has explored optimizations of the classic Hierarchical Scheme [13, 14], including the use of cuckoo hashing [17, 19, 20] and Bloom filters [16]; used incautiously, these optimizations can leak information [20]. This work has steadily improved both the asymptotics, constants, and security of earlier schemes, to the point of achieving $O(\log^2 N/\log\log N)$ overhead/access, for a data set with N items [20].

Traditional ORAM [14] assumes limited – $O(1)$ – processor storage, but a few efforts have improved response times by assuming more storage, e.g., $O(\sqrt{N})$, on the processor [18, 35]. Given the capabilities of modern coprocessors and the size of data center data sets (§4), schemes requiring $O(\sqrt{N})$ trusted storage are infeasible. They would require gigabytes of coprocessor memory, when even high-end secure coprocessors have only 32 MB of memory and low-end ones have only 16 KB.

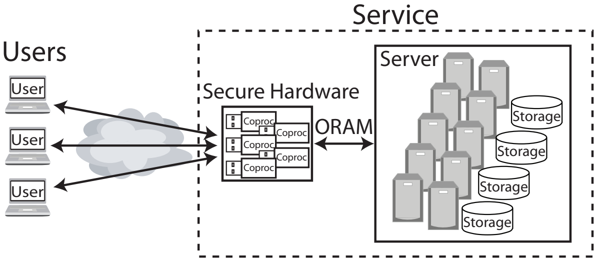


Figure 1: **Private Data Access Overview.** *To read and/or write data, a user establishes a secure channel to a secure coprocessor in the service's data center. Along with other coprocessors, the coprocessor uses an ORAM algorithm to retrieve or update a block without revealing to the service which block was accessed. It returns the result via the secure channel.*

Very recent work considers practical aspects of ORAM [18, 36–38], but they perform all operations between the client and the data center (rather than using a hardware proxy) and/or run code on the main CPU, rather than on actual secure hardware. Concurrent work also adds parallelism [36], but only across multiple requests, rather than within a single request.

For use in a data-center environment, we selected Shi et al.'s Binary Tree scheme [15] (§5.1). It suits our goals because it has $O(\log^3 N)$ *worst-case* access time, its binary tree structure lends itself to parallelization, and its algorithm uses simple primitives that lend themselves to implementation on a coprocessor. It also uses $O(1)$ storage, making it suited to our limited secure coprocessors.

3 Problem Definition

3.1 Execution Model

At a high level, remote users access data privately by indirecting their requests through secure coprocessors running in the service's data center (Fig.1). A coprocessor uses standard attestation techniques [39] to convince those users that it is a real secure coprocessor running the expected code. Collectively, using the data center's servers as untrusted storage, the secure coprocessors implement an ORAM algorithm, allowing them to answer requests without leaking information to the server.

In contrast to having the users execute the ORAM algorithm directly, this model keeps the bandwidth-intensive ORAM protocol within the data center network. It also allows the users to share a single ORAM, rather than creating an ORAM per user or forcing the users to share a secret key.

In this work, we assume each coprocessor has limited internal trusted storage that can indefinitely store a few keys and a small amount of state. The coprocessors are distinct from the *server*, which is the untrusted component that handles storage and communication. The server may run on multiple processors and/or machines to achieve parallelism and fault tolerance.

3.2 Threat Model

We assume the attacker physically controls the service's data center and can run arbitrary code on the servers. The attacker can also submit known queries to the service and observe the service's internal operations in response. However, we assume he cannot violate the tamper-responding secure coprocessors. We assume this not because secure coprocessors are absolutely inviolate, but because they present a high barrier. By defending against the large class of attackers who cannot subvert them, we substantially increase security.

We do not consider denial-of-service attacks, since the attacker can always power off the service or sever the network connection. We also do not consider attacks that could be launched by a standard network attacker; e.g., the attacker is able to observe that a particular IP address has submitted three requests today. If desired, users can mitigate this risk via standard techniques [10].

3.3 Strawman Solutions

Used naïvely, parallelism can be problematic.

Replication. One natural approach to using a collection of secure coprocessors is replicated execution, i.e., to have each secure coprocessor maintain an independent copy of the ORAM data structure. Such a scheme would trivially improve response throughput, but not latency. Unfortunately, this also increases storage requirements enormously. For a database of N items, ORAM schemes typically require the untrusted server to store between $O(N)$ and $O(N \log N)$ encrypted blocks. The encryption key is unique to the ORAM scheme, meaning that each replica of the ORAM scheme would require its own freshly encrypted copy of the original database. While storage is indeed growing cheaper, existing data sets are already hundreds or thousands of terabytes (§4.1), so replicating them tens of thousands of times is impractical. It is also unclear how this would support writes.

Distributed Caches. A more practical approach would have all the secure coprocessors share a key and operate on the same encrypted data set. Unfortunately, dealing with writes as well as reads is complicated, and even handling only reads may leak information about whether two requests are for the same address. Imagine two requests arrive for item v, each routed to a different coprocessor. If neither coprocessor has previously fetched v, then neither one will find it in their cache, and hence both will request v from the server. From this, the server will observe that two different requests were actually for the same block, undermining ORAM's obfuscation.

Typical secure coprocessors are I/O-bound (§4.2), so coordinating coprocessors via explicit messages is too expensive. Instead, we require new protocols to synchronize the coprocessors and keep the ORAM secure.

Data Set	# of Blocks	Block Size
Map Tiles	2^{35}	10 KB
Twitter Tweets	2^{35}	0.14 KB
Facebook Images	2^{36}	10 KB
Flickr Photos	2^{32}	5 MB

Table 1: **Data-Center-Scale Data Sets.** Approximate sizes.

4 Operating Constraints

The main practical challenges of using ORAM are the large size of real data sets used by real web services (§4.1), and the limitations of secure hardware (§4.2).

4.1 Big Data

Modern web services expose vast data sets via public interfaces; we describe several examples (Table 1).

Map Tiles. Many location-based services ultimately plot a user's location, or her friend's, on a map. Even if the exchange of location data is strongly protected [40], retrieving map information may leak the location data. Thus, a privacy-preserving map service must also ensure the obliviousness of a user's tile requests.

To characterize a real-world map service, we sampled Bing Maps tiles on "road" view. This service uses 256 × 256-pixel tiles organized by levels of detail, also known as zoom levels. The number and size of tiles varies by level, but about half ($\sim 2^{35}$) are at level 19. The maximum tile size at that level is ∼10 KB.

Twitter. Although Twitter posts or "tweets" are public, a user may wish to conceal which accounts she follows or which tweets she is interested in. According to public reports [41], Twitter currently receives one billion tweets every week. Extrapolating from previously published metrics [42], we estimate that Twitter has received approximately 2^{35} tweets throughout its history. Each tweet is 140 bytes or less.

Facebook Images. Social networks raise many privacy concerns, but even if they migrate to more decentralized and privacy-enhancing platforms [6], data retrieval will still threaten privacy unless it hides access patterns.

An illustrative data set is Facebook's photo sharing site, with more than 65 billion photographs consuming 20 petabytes [43]. Each photograph is stored in four image sizes. Extrapolating from published numbers and images sampled from Facebook, we estimate that these sizes consume an average of 3 KB, 10 KB, 18 KB, and 71 KB respectively. A site deploying ORAM for these images would likely use a separate ORAM structure for each size. Since 84.4% of image requests are for small images [43], we choose this image size as representative.

Flickr Photos. Flickr is another popular photo sharing site with over six billion photos as of August 2011 [44]. Like Facebook, it stores versions of each photo in multiple sizes; unlike Facebook, it also makes the original image available. Consumer-grade cameras produce images in the 3–14 MB range, so we arbitrarily choose 5 MB to represent data sets with larger blocks.

	Infineon SLE 88	IBM 4764
CPU	66 MHz	266 MHz
Memory	16 KB	32 MB
I/O	12 KB/s	9.85 MB/s
3DES, 1 KB	73 KB/s	1.08 MB/s
SHA-1, 1 KB	155 KB/s	1.42 MB/s
Cost	\$4	\$8,000

Table 2: **Secure Coprocessor Performance.** *Specifications and measured performance of representative coprocessors.*

4.2 Secure Hardware Performance

We selected a representative from each end of the coprocessor spectrum and performed microbenchmarks to characterize their performance (Table 2). Both devices provide cryptographic accelerators, internal key storage, active protection against physical tampering, and the ability to attest to the code they run [39].

At the low end, the Infineon SLE 88 [45] is a small but surprisingly powerful chip often found in smart cards, pay TV boxes, and various military applications. For use in a data center, it can be packaged in a USB dongle. We use the CFX4001P cards, which come with 400 KB of EEPROM, 16 KB of RAM, and a 66 MHz CPU. The chip's design is EAL5+ certified, and the physical packaging is certified at FIPS 140-2 level 3, meaning that it has a high probability of detecting and responding to physical attacks. It includes sensors to detect voltage and temperature irregularities, and it draws power across a capacitor to frustrate power analysis. Its biggest appeal is its price, \$4, making it feasible to add one to every computer in a data center.

In contrast, the IBM 4764 [46] is a high-end secure coprocessor. Each contains a 266 MHz PowerPC 405 CPU, with 32 MB of RAM and much faster I/O and cryptographic processing. The 4764 is certified at FIPS 140-2 level 4, the highest possible, meaning it includes strong protections that detect physical tampering and respond by zeroing its secrets.

5 Scaling Oblivious RAM to Data Centers

In this section, we describe our scheme for achieving a highly parallel ORAM service. We start by describing the Binary Tree algorithm [15], then show how to restructure it to enable parallelism, fault tolerance, and resilience to a malicious server. Proofs of correctness and security appear in Appendices A–D.

5.1 Background: The Binary Tree Algorithm

We first describe the Binary Tree ORAM algorithm of Shi et al. [15] without any of our modifications. Figure 2 summarizes the notation, Figure 3 contains pseudocode, and Figure 4 depicts a high-level overview.

Symbol	Meaning
N	# of block addresses
B	Block size in bits
M	Maximum # of operations on the ORAM
δ	Probability of ORAM failure
R	# of stages, typically ≈ 4 for our data sets
c	# of designators that can fit into a block

Figure 2: **Notation Summary.**

5.1.1 Functionality

We assume that the data blocks stored in the ORAM have virtual addresses ranging from 1 to N. We further assume that all blocks have size B bits. We assume there will be a maximum of M operations, and that we can tolerate a failure probability of δ during that sequence of operations. §5.5 presents a method we devised to remove this assumption of a bounded number of operations.

Users can perform reads or writes. A read takes an address and returns the block with that address. A write takes an address and new block, and overwrites the old block at that address. To prevent the server from distinguishing between reads and writes, we treat them identically; this requires an ignored input block for reads and an ignored output block for writes.

5.1.2 Operation

The Binary Tree algorithm, as illustrated in Figure 4, uses a binary tree with N nodes called *buckets*. Each bucket contains $O(\log N)$ *entries*, each of which contains an address and a block. Each bucket is treated as a "trivial ORAM"; it is accessed by examining all entries and reading or updating the one with the desired address.

To track an entry's location in the tree, the coprocessor maintains a mapping from addresses to *designators*. A block will always be in an entry of a bucket on the path from the root to the leaf indicated by the designator.

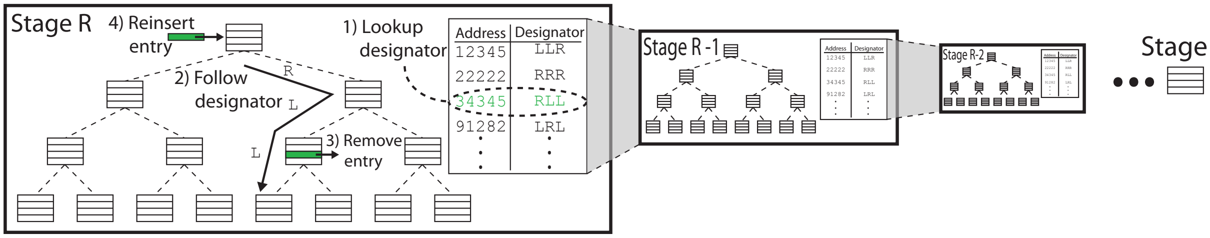
When a data entry is written to the ORAM, we insert it into the root bucket of the binary tree. To prevent buckets from overflowing, on each request, we randomly select two buckets from each level of the tree, for a total of $O(\log N)$ buckets, to *evict* from (`EvictStage` in Figure 3). To evict from a bucket, we remove a valid entry from it and add that entry to the bucket's child, along the path toward the evicted designator's leaf bucket. If no empty space exists in that child, the ORAM algorithm fails, so bucket sizes are chosen so that the probability this ever happens is $\leq \delta$.

When a user requests a read or write at an address (`ORequest`), we look up the address's designator in a table that maps addresses to designators. We then read all buckets in the tree along the path between the root and this designator (`LookUpAndRemove`). When we find the entry, we remove it from its current bucket and rewrite it at the top of the tree using a new designator. Thus, repeated reads for the same entry will produce different lookup paths through the tree.

```
ORequest ($\overline{\text{Address}}$, $\overline{\text{IsWrite}}$, $\overline{\text{NewBlock}}$):
  For each Stage from 1 to R:
    If Stage = 1:   $\overline{\text{Block}}$ ← the sole stage-1 block
    Else:
      EvictStage(Stage)
      $\overline{\text{Block}}$ ← LookUpAndRemove($\overline{\text{Address}}$, Stage,
                                     Designator)
    If Stage = $R$:
      If $\overline{\text{IsWrite}}$:   $\overline{\text{Block}}$ ← $\overline{\text{NewBlock}}$
    Else:
     $\overline{\text{CurDes}}$ ← designator at $\overline{\text{Address}}$'s offset in $\overline{\text{Block}}$
     Randomly update des. at $\overline{\text{Address}}$'s offset in $\overline{\text{Block}}$
     Designator ← $\overline{\text{CurDes}}$ if valid, else random
    Insert $\overline{\text{Block}}$ into the root bucket for Stage
  Return $\overline{\text{Block}}$

LookUpAndRemove($\overline{\text{Address}}$, Stage, Designator):
  $\overline{\text{Result}}$ ← 0
  For each Bucket along tree path toward Designator:
    For each entry index in Bucket:
      Read the entry from storage into $\overline{\text{Entry}}$
      If $\overline{\text{Entry}}$'s address field matches $\overline{\text{Address}}$:
        $\overline{\text{Result}}$ ← $\overline{\text{Entry}}$'s contents field
        $\overline{\text{Entry}}$ ← 0
      Re-encrypt $\overline{\text{Entry}}$ and write it to storage
  Return $\overline{\text{Result}}$

EvictStage(Stage)
  Compute the set of buckets to evict BucketsToEvict
  For each Bucket in BucketsToEvict:
   $\overline{\text{SelectedIndex}}$ ← index of a valid entry in Bucket
   $\overline{\text{EvictedEntry}}$ ← entry $\overline{\text{SelectedIndex}}$ of Bucket
   Invalidate entry $\overline{\text{SelectedIndex}}$ of Bucket
   Re-encrypt other entries of Bucket to obscure access

   $\overline{\text{Designator}}$ ← designator in $\overline{\text{EvictedEntry}}$
   $\overline{\text{Child}}$ ← Bucket's child on path toward $\overline{\text{Designator}}$
   $\overline{\text{SelectedIndex}}$ ← index of an empty entry in $\overline{\text{Child}}$
   Entry $\overline{\text{SelectedIndex}}$ of $\overline{\text{Child}}$ ← $\overline{\text{EvictedEntry}}$
   Re-encrypt all entries of Bucket's children to obscure
```

Figure 3: *Basic Pseudocode for ORAM operations. Overlined values must be hidden from the server.*

The table mapping addresses to designators must itself be accessed obliviously. However, it contains $O(N)$ mappings, making it far too large to store locally. Fortunately, each mapping is tiny, about $\log_2 N$ bits, so the count c of mappings that fit in a block is large. Thus, we can recursively apply the same ORAM scheme to this smaller collection of blocks. Each ORAM, or *stage*, will be progressively smaller, and the Rth stage where $R \approx \log_c N$ will be a single block. As a result, each access performs $R-1$ lookups on increasingly larger ORAMs to determine the designator for the requested entry. The entry is then retrieved from, or written to, stage R. R is generally quite small; indeed, for the data sets from §4.1, $R \approx 4$.



Address	Designator
12345	LLR
22222	RRR
34345	RLL
91282	LRL
⋮	⋮

Figure 4: **Overview of the Binary Tree Algorithm.** *Each of the N buckets in the binary tree consists of $O(\log N)$ entries. To find the entry for a given address, we look up the address's designator, and follow the path through the tree dictated by that designator. For each bucket on the path, we read all of the entries in that bucket. This continues along the entire path, even if we find the address we're looking for, to hide the entry's actual location from the server. After reading or writing the entry, we remove it from wherever it was found, assign it a new designator, and reinsert it into the top-level bucket. Since the address-to-designator table is large, it is itself stored in a recursive version of the ORAM structure. Each stage is smaller than the one it stores designators for.*

The following example illustrates this recursion. Suppose an ORAM has 1,000 addresses and each block holds 10 designators, so the number of stages is $R = 4$. If the user asks for address 789, we first fetch the sole block of the stage-1 ORAM and extract the designator at offset 7. This is needed to find a block in the stage-2 ORAM. This stage-2 block contains, at offset 8, the designator needed to find a block in the stage-3 ORAM. Finally, this stage-3 block contains, at offset 9, the designator needed to find the user's requested block in stage 4.

5.1.3 Storage

The server provides storage to support the ORAM service. Since the server is distrusted, coprocessors encrypt all sensitive data using a symmetric key with an IND-CPA secure encryption scheme before storing it with the server. The IND-CPA property ensures that the server cannot learn anything about the underlying plaintext, not even whether it has been modified by the coprocessor. Each encrypted item has a name, like *"7th entry in bucket 12 at depth 6 of stage 3,"* so that coprocessors can tell the server which item to store or retrieve.

5.2 Parallelizing ORAM for the Data Center

5.2.1 Initialization

To parallelize the Binary Tree algorithm, we need to be able to form multiple coprocessors into a coordinated, mutually-trusting group. We do this by letting any participating coprocessor C vet and induct any other coprocessor C_{new} that wishes to join the service.

To initiate the service, the server connects to a coprocessor and asks it to create a new ORAM of size N and block size B. The coprocessor generates fresh key material and outputs a public key, PublicKey.

To join the service, C_{new} asks the server for PublicKey. Via attestation [39, 47, 48], C and C_{new} convince each other that they are both running the correct code in a secure environment. C_{new} then uses PublicKey to create a secure channel to C, and C transmits the service's secret key material to C_{new}.

5.2.2 User Requests

A user initiates secure communication with the ORAM service by requesting a standard attestation from one of the secure coprocessors [39, 48]. The attestation certifies that PublicKey is held only by true secure coprocessors running the ORAM code. Using PublicKey, the user creates a secure channel to a coprocessor. She uses this channel to convey the operation she wishes to perform, without revealing it to the server. The coprocessor coordinates with the other secure coprocessors in the data center to perform the Binary Tree algorithm in parallel, as described below. Eventually, the coprocessor returns the result over the secure channel.

5.2.3 Parallelism Goals

Our primary aim is to minimize coprocessor reads and writes of large blocks on the critical path. We also use parallelism to mitigate computationally taxing steps. We only worry about expensive computations, such as hashing and encryption. Thus, we treat as $O(1)$ any operation that is technically $O(\log N)$ but actually fast, like an XOR of two $\log_2(N)$-bit numbers. To distinguish expensive coprocessor operations from relatively cheap server operations, we use $O_C(\cdot)$ to summarize coprocessor bandwidth and CPU costs, and $O_S(\cdot)$ for server overhead.

The main limit to our ability to parallelize is the iteration of lookups through the $R \approx \log_c N$ stages. Each iteration produces the designator for the next, so we cannot perform them in parallel. Fortunately, since c is large, R is typically small. For our data sets, and for most data sets we can envision, it is five or less.

The primary bottleneck of the algorithm is the R entry lookups, each of which reads B-size entries from the server in two nested loops: one over the $O(\log N)$ buckets in the path, and one over the $O(\log N)$ entries in each bucket. Thus, the overhead is $O_C(RB\log^2 N)$. This is substantial given how slowly coprocessors operate.

Below, we show how to employ the parallelism of $O(\log^2 N)$ coprocessors to reduce the time, including bandwidth use, to $O_C(RB)$. We find that, for the data sets we consider and expect to see in practice, this allows us to achieve low latency (§7).

5.2.4 Parallelizing Bucket Accesses

We show how to access entries in a bucket in $O_C(B)$.

Parallelizing entry updates is straightforward since there is no dependency between entries in a bucket. By assigning one coprocessor to each of the $O(\log N)$ entry indices, we can complete all iterations in the time it takes to do one. The dominant cost for this is that of reading, decrypting, encrypting, and writing an entry: $O_C(B)$.

It is more difficult to parallelize entry reads. If we assign one coprocessor to each entry index examined, they must all cooperate to collectively produce the output. Fortunately, we know that exactly one of the coprocessors has something to contribute to this computation, since each is assigned a different index but only the one assigned index i has anything to contribute to the computation. Thus, the output can be computed by XOR'ing the coprocessors' respective outputs. In §5.2.7, we present an algorithm called *oblivious aggregation* for efficiently (in $O_C(B)$ time) and securely computing such a collective XOR.

Finding a valid or invalid entry index in a bucket uses a lot of coprocessor bandwidth, since it requires reading all of the bucket's entries. Thus, we separate this small, frequently-accessed validity information from the large $O(B)$-size entries containing them. Specifically, we use a *validity vector*: a vector of bits, one per entry, indicating if that entry is valid. Naturally, we must also keep this vector current whenever we change a bucket.

5.2.5 Parallelizing Lookups

To look for an entry (`LookUpAndRemove`), we must look at all $O(\log N)$ entries in all $O(\log N)$ buckets on the path to the designator. This takes $O_C(RB\log^2(N))$ time. But, if we access all entries in parallel on $O(\log^2(N))$ coprocessors, we only use $O_C(B)$ bandwidth $R-1$ times. The challenge is that these coprocessors must cooperate to compute the lookup result.

Fortunately, for each of these computations, at most one of the coprocessors will have something to contribute to it. This is because an invariant of the algorithm is that the same virtual address appears in at most one bucket of the tree [15]. This is ensured by preceding any write to address v with a removal of the entry with address v from the tree. Thus, the output (Result) can be computed by XOR'ing the coprocessors' respective contributions, at most one of which is nonzero. This means we can use oblivious aggregation (§5.2.7) to effect this computation.

A minor wrinkle is that, as we will see, oblivious aggregation requires the server to XOR together $O(\log^2 N)$ values. Fortunately, the commutativity of XOR makes it straightforward to parallelize using a modest number of untrusted server processors. Each of $O(\log N)$ server processors can XOR $O(\log N)$ values in parallel, then one processor can XOR those processors' results; this all takes $O_S(RB\log N)$ time. If increased speed is desired, each of $O(\log^2 N)$ server processors can do it in $O_S(RB\log\log N)$ time. Since these XORs take place on fast server processors at GB/s, orders of magnitude faster than coprocessor writes, the time spent on this is trivial.

5.2.6 Parallelizing Eviction

Finally, we consider entry eviction (`EvictStage`). Naïvely, this requires iterating serially through the tree. For each bucket selected for eviction, we find a valid entry, remove it, find an empty space in its child, and update both children so the server cannot tell which child receives the entry. Since we evict from $O(\log N)$ buckets at each stage and evict to twice that many, this takes $O_C(RB\log^2 N)$ time.

Fortunately, we can parallelize eviction by assigning a coprocessor to each of the involved entries. We have already seen how to parallelize access to a bucket across one entry per coprocessor. Furthermore, eviction operations for buckets are independent of each other, as long as we first do all eviction from buckets and then do all eviction to buckets. As a consequence, eviction takes $O_C(RB)$ time with $O(\log^2 N)$ coprocessors.

In addition to the parallelism within eviction, we observe that eviction and lookup can themselves be performed in parallel, given enough coprocessors. This can be done safely since evict only moves each entry along the path specified by its designator, while lookup examines all buckets on that path. Thus, eviction cannot change whether a lookup will succeed.

5.2.7 Oblivious Aggregation

Sometimes our algorithm requires q secure coprocessors to aggregate their individual values, without revealing these values to the untrusted server. More concretely, each secure coprocessor m knows some private value $v_m \in \mathcal{V}$, and they collectively need to output the combined value $S \leftarrow \bigoplus_{m=1}^{q} v_m$ without revealing any particular v_m. We could accomplish this by having each secure coprocessor encrypt and output its value, and then have one secure coprocessor read in, decrypt, and XOR all of the values together, but this would take $O_C(q)$ I/O operations, which are quite expensive for the secure coprocessors (§4.2). If we use the protocol below, we only need a single I/O operation from each secure coprocessor. The server still does $O_S(q)$ computation, but the constants are so small that this is essentially negligible.

In our protocol, we assume that the secure coprocessors are given a fresh nonce j and that each secure coprocessor knows its distinct index 1 through q. We also assume the secure coprocessors share a key for a cryptographically secure pseudorandom function PRF.

Protocol: Given its secret value $v_m \in \mathcal{V}$, shared key K, fresh nonce j, and the number of secure coprocessors q, each secure coprocessor m computes

$$x_m = \mathsf{PRF}_K(j \parallel m) \oplus \mathsf{PRF}_K(j \parallel (m+1 \bmod q))$$

It then outputs $z_m = x_m \oplus v_m$. From these, the untrusted server computes and outputs $Z = \bigoplus_{m=1}^{q} z_m$.

Appendix A proves the correctness and security of this protocol.

5.3 Dealing with Malicious Servers

Thus far, we have assumed an honest-but-curious server. But, if we distrust the server enough to use ORAM to hide our data access patterns, it seems sensible to treat the server as fully malicious. Since the server acts as a relay between coprocessors, it has four broad classes of attack available: attacks on data secrecy, integrity, freshness, and availability.

Attacks on *data secrecy*, are, at a high level, handled by the ORAM protocol itself, and, at a low level, prevented by encrypting all secret values with an IND-CPA secure scheme. Combined with a MAC, this protects data secrecy against a malicious server [49]. *Data integrity* can largely be addressed by having the coprocessors MAC their outputs and verify MACs on their inputs. The oblivious aggregation protocol complicates this, however, since the server combines the coprocessors' output, but cannot produce a MAC on the result (§5.3.1). *Data freshness* is generally complicated in ORAM schemes, since, for efficiency reasons, each request touches only a portion of the data structure (§5.3.2). Finally, we ignore *data-availability* or denial-of-service attacks, since they are impossible to prevent; the coprocessors are entirely dependent on the server for communication.

Overall, we defend against malicious servers using only $O_C(RB + R\log N)$ time per request, compared to $O_C(RB)$ for honest-but-curious servers. We require no more coprocessors to perform the necessary validations than are needed to parallelize the basic algorithm.

5.3.1 Ensuring Data Integrity

As mentioned above, most integrity attacks can be prevented by having each coprocessor MAC its outputs and check the MAC on its inputs. Coprocessors also check data from the server, e.g., to check that a given bucket is indeed scheduled for eviction based on the current request number.

The use of oblivious aggregation complicates integrity checks, since the server computes the final output Z and clearly cannot produce a corresponding MAC. Thus, we ensure that Z is always self-attesting; i.e., it consists of a statement and a MAC of that statement's hash.

5.3.2 Ensuring Data Freshness

The attacker can learn inappropriate information if allowed to present stale state to a coprocessor. By replaying an old state with the same input as before, he may learn something from the differences in the coprocessors' output. By replaying an old state with a new input, he may learn something from the differences and similarities between how the coprocessor handles the two inputs.

We prevent attacks in the first class by making coprocessors stateless and deterministic. This ensures that given the same state with the same input, a coprocessor will always produce the same output. An exception is that encryption uses randomness, but since our encryption scheme is IND-CPA secure, the server gains no information by making us encrypt the same content repeatedly with different randomness.

We prevent attacks in the second class by uniquely numbering each request and associating each state with one request, as we will now describe. This way, given the same state with two different requests, coprocessors will only act on the request associated with that state.

UNIQUELY NUMBERING REQUESTS. To ensure each request receives a unique number, we use a master coprocessor. §5.4 shows how to replicate the master for fault tolerance, but for simplicity here we treat it as a single coprocessor. When the user initially submits a request to a coprocessor, the coprocessor encrypts it and provides it to the server, which gives it to the master. The master keeps an internal counter of how many requests have occurred thus far. When a new request arrives, it assigns it the next request number and increments the counter. The master binds the request to the assigned number by outputting a MAC of the two values.

BINDING STATES TO REQUESTS. We use an authenticated data structure to bind each state of the ORAM data structure to the request that produced it. Thus, coprocessors will only act on state for request $r+1$ if the state is bound to request r. Giving a coprocessor state r with any other request will produce an error.

Binding the entire ORAM structure to a request number is difficult since it is has enormous size, $O(BN\log N)$. To cope with this scale, we use a collection of Merkle trees [50]. A Merkle tree is a tree of hashes such that each leaf is the hash of a component of a data structure, and each parent is the hash of the concatenation of its children. A MAC using the root hash is equivalent to a MAC using a hash of all data structure components.

We use one Merkle tree for each combination of stage, tree half (left or right), and depth. We have a separate Merkle tree for each combination of half and depth because, as we will see, it makes it easy to quickly evaluate whether an entire subtree is unchanged by an operation. This makes it tractable to generate a new root hash after

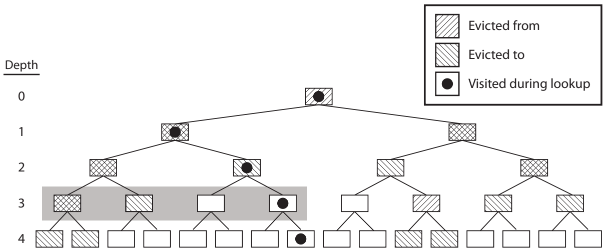


Figure 5: **Accessed Buckets.** *In each half of each level of the bucket tree, at most four buckets are accessed during a request: one evictor, two evictees that are children of evictors, and one that is accessed by the lookup for the specified address. Sometimes these overlap, particularly at the low depths depicted in this figure; e.g., in the left half of depth 3 (shaded), one bucket is an evictor and evictee, so only three buckets are accessed.*

each operation, since large swaths of the Merkle tree that are unchanged do not have to be touched.

Each Merkle-tree leaf is the hash of a bucket's entry, with a bucket's validity vector treated as the zeroth entry. Intermediate hashes above the leaves represent continuous ranges of entries within the same bucket, and the topmost of these represent entire buckets. Above that, Merkle hashes represent continuous ranges of bucket indices within the same depth. The root hash represents all entries in all buckets, but only in one half of the tree, at a given depth, and in a given stage.

Because of this structure, it is possible for a coprocessor to quickly evaluate whether a Merkle hash is unchanged by an operation. Each hash corresponds to only a single side and depth, and a certain bucket index range within that depth. A coprocessor can be told, using only $O(1)$ bandwidth, what bucket index was looked up at that side and depth, and it can compute in $O(1)$ time which buckets were evicted from a given side and depth. Thus, a coprocessor can quickly determine whether anything could have changed in the subtree summarized by the Merkle hash. If not, it can treat an attestation of the hash's value after request r as an attestation of the hash's value after request $r+1$.

We now analyze the time to read or update the Merkle trees. We have seen that a coprocessor can access a Merkle node in $O_C(1)$ time. For any Merkle depth d, all Merkle nodes can be handled in parallel, even nodes from different Merkle trees and even different stages. This is because, as we discuss further in §5.5, we can do eviction for all stages at once and even overlap eviction with lookup. Since each Merkle tree has Merkle depth $O(\log N)$, the total time to update all the Merkle trees is $O_C(\log N)$.

Next, we analyze the number of coprocessors needed to achieve this time. Each Merkle tree corresponds only to buckets from one side and depth, and thus contains at most four buckets that can change. This is because at each bucket depth and half there is at most one evictor, two evictees, and one bucket used for lookup (Fig. 5). Since each bucket contains $O(\log^2 N)$ entries, and there are R stages, there are at most $O(R\log^2 N)$ nodes at each Merkle depth that change. Thus, we need at most $O(R\log^2 N)$ coprocessors for maximum parallelism.

5.4 Fault Tolerance

Since it is fairly well understood how to make untrusted components fault-tolerant through replication, we discuss only how to tolerate coprocessor failures.

Our system is designed so all of the coprocessors, except the master coprocessor, keep no long-term state beyond N, B, UHFSeed, and keys (§5.3.2). Thus, if a nonmaster fails, its functionality can be duplicated by having a freshly initialized coprocessor join the ORAM service.

To deal with the failure of the master coprocessor, the master should actually consist of $2f+1$ cooperating *master instances*. Here, f is the number of master coprocessor faults we want to tolerate. A coprocessor becomes a master instance by randomly choosing a fresh, globally-unique instance identifier and storing this in nonvolatile storage, alongside a counter set to zero. The counter's value indicates the highest request number the instance has assigned using this identifier. The global uniqueness ensures that if the coprocessor loses its state and creates a new instance, the new instance will not be confused with the old one. Thus, the counter associated with an instance will never roll back.

A master instance will assign, via attestation, any number higher than its current counter to any request the server asks it to. A request is considered definitively assigned a number when a quorum of the instances, i.e., at least $f+1$ of them, have assigned it the same number.

To cope with master instance failures, we replace the master instances periodically, say once a day, via *delegation*. To begin delegation, the server selects a new set of master instances, and asks the existing masters to delegate their authority to the new group starting with the next request number r. A master performs this delegation if its counter is less than r. It delegates its authority by deleting its master instance and outputting an attestation of its delegation that includes r and the new set of master instances. Once the server collects such attestations from a quorum of the old master instances, it presents them as proof that the new master instances are responsible for requests r and beyond. Appendix B contains a proof that delegation can never cause two requests to be assigned the same request number.

5.5 Recommended Optimizations

In this subsection, we will describe some additional optimizations we recommend. To describe them fully, we will need the following additional terminology.

We denote by N_s the number of addresses served by the stage-s ORAM, so $N_R = N$. Each stage s is responsible for storing the designators for the N_{s+1} addresses in stage $s+1$. Each designator takes $\leq \lceil \log_2 N \rceil + 1$ bits, so a block can hold at least c pointers, where $c = \left\lfloor \frac{B}{\lceil \log_2 N \rceil + 1} \right\rfloor$. Thus, we use $N_s = \lceil N_{s+1}/c \rceil$.

The storage for ORAM stage s consists logically of a binary tree with depth $D_s = \lceil \log_2(N_s) \rceil$. Each node in this tree is a bucket with E_s entries. To keep the probability of overflow bounded by δ, Shi et al. demonstrate $E_s = O(\log_2(MN/\delta))$ is sufficient [15]. The root and depth-1 buckets need only one entry each since they are evicted on every operation.

Shortening the Critical Path. A critical bottleneck is the need to iterate through all the stages, since each stage must look up a designator before the next stage can proceed. However, we can arrange for stage s to output the designator needed by stage $s-1$ as soon as possible, thereby overlapping the rest of stage s with subsequent stages. A simple way to do this is to break `LookUp-AndRemove` into a lookup phase and a remove phase, and start the next stage after the lookup phase completes.

We can reduce the critical path even further by having the lookup phase first output just the part of the block needed by the next phase. By encrypting blocks with counter mode, a coprocessor can quickly decrypt and output just the part that contains the designator needed by the next phase. It will later have to decrypt the entire block, but this can be overlapped with subsequent stages.

Removing Eviction From the Critical Path. A stage can do eviction before its lookup designator is known. All that is needed is the block address, so that the eviction-based update can invalidate any entry it finds with that address; unlike the designators, this address is known at the beginning of `ORequest`. Thus, any extra coprocessors not needed by the critical path can be assigned to perform steps of `EvictStage` for any or all stages, even stages that have not yet begun `LookUp-AndRemove`.

Verifying Evicted Buckets. Since we cannot trust the server to accurately report which buckets are scheduled for eviction, we need each coprocessor to verify these choices. Furthermore, as discussed in §5.3.2, it is useful for a coprocessor updating a Merkle tree to quickly compute the set of buckets evicted from at each side and depth. Therefore, we use the following technique to efficiently compute the evicted buckets.

A coprocessor or server can quickly compute the indices of the depth-d buckets to evict during request r as $\mathsf{UHF}(\mathsf{UHFSeed} \parallel d \parallel r \parallel 0) \pmod{2^{d-1}}$ and $\mathsf{UHF}(\mathsf{UHFSeed} \parallel d \parallel r \parallel 1) \pmod{2^{d-1}} + 2^{d-1}$. This method is not securely random in that the server can know, even in advance, what choices will be made. Fortunately, hiding this information is not important to the scheme's security properties.

Note that this always selects one bucket from the left half of the tree and one bucket from the half. This does not impact the overflow analysis since it maintains the same probability of eviction for each bucket. We do this so a coprocessor working on a bucket can verify whether it is scheduled for eviction quickly, using one UHF evaluation; it can also verify if its parent is scheduled for eviction with another UHF evaluation.

Smaller Non-leaf Buckets. By improving on the overflow analysis of Shi et al. [15], we can use fewer entries for non-leaf buckets than are necessary in leaf buckets: $\log_2(M(D_s-2)(R-1)/\delta)$ instead of $\log_2(MN_s(R-1)/\delta)$. This reduces the number of entries per non-leaf bucket by almost $\log_2 N_s$, and half of all buckets are non-leaves. The reason we can use fewer entries in non-leaves is that overflows of non-leaf buckets can only happen as a consequence of eviction, and only $2M(D_s-2)$ evictions ever happen; for a full proof, see Appendix C.

In practical terms, workloads we consider require about 19% fewer entries per non-leaf due to this optimization. Although non-leaves only account for half of all buckets, they account for nearly all of the buckets accessed in any ORAM operation. Thus, the effect on latency is significant, as we will see in §7.6.

Packing Designators. Designators are $D_s + 1$ bits long. For a stage $s < R$, $D_s = \lceil \log_2(N_s) \rceil < \lceil \log_2(N) \rceil$, so designators take fewer than $\lceil \log_2 N \rceil + 1$ bits. This means a block can hold much more than c such designators; it can hold $C_s = \lfloor \frac{B}{D_s+1} \rfloor$ of them. Thus, we actually only need $N_s = \lceil N_{s+1}/C_s \rceil$ blocks in stage s. If we pack designators into blocks in this more efficient manner, all the N_s values for $s < R-1$ get smaller, thereby saving time. In some cases it may even reduce R by one, which would save even more time.

Extending Service Lifetime. If M might be an underestimate of the number of operations the ORAM will have to perform, it is possible to plan for this eventuality, as follows. At the beginning, we use one additional entry per bucket. After each operation numbered $(2^i - 1)M$, where i is a positive integer, we add an additional two entries per bucket. This allows indefinite operation, at the cost of a slight increase in the number of entries per bucket: If we underestimate M by a multiplicative factor of u, we use $\log_2(u+1)$ more entries than necessary. The additional entry at the start ensures that the probability of overflow in the first M operations is $\delta/2$ instead of δ. Every time we add two entries, we divide the subsequent overflow probability by four, allowing the ORAM to perform twice as many operations with half the probability of overflow. Thus, the probability of overflow ever happening is $\leq (1/2)\delta + (1/4)\delta + (1/8)\delta + \cdots = \delta$.

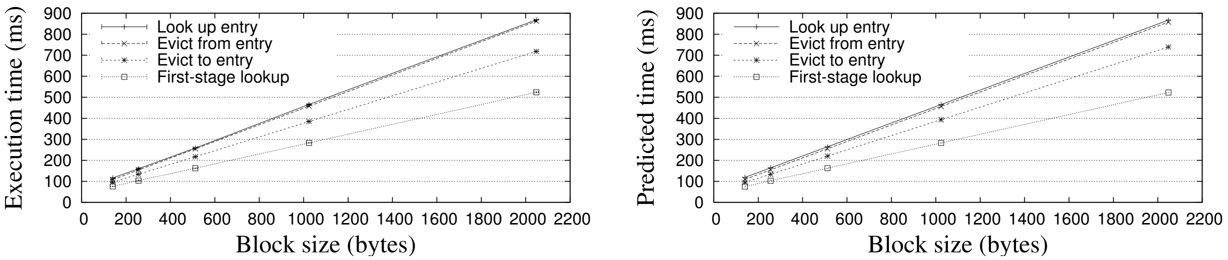


Figure 6: **SLE 88 Performance.** *The effect of block size on the time to perform the most performance-impacting operations in our ORAM algorithm. The left side shows actual measured performance, and the right side shows our simulator's predictions. Error bars, generally too small to see, indicate 95% confidence intervals about the mean of 50 trials.*

6 Implementation

Our implementation of Shroud consists of two components: the coprocessor and the server.

The coprocessor component runs on Infineon SLE 88 cards. The implementation includes all aspects of our algorithm, including protection from malicious servers as described in §5.3. It is 2,898 lines of C, as measured by SLOCCount [51]. Our code uses 3DES for encryption, since the smart card supports it with hardware acceleration. It uses SHA-1 as its hash function, HMAC for a MAC, and the NH hash from UMAC [52] as the UHF.

The server component consists of a worker, a scheduler, and a file server. A worker attaches to a coprocessor and sends it commands. The scheduler parcels out work among several workers to complete a user's request. The file server provides access to items by name and also facilitates communication between workers. The server component is 5,000 lines of C#; 932 of these lines are the scheduler. We have not yet implemented user interaction or fault tolerance.

To evaluate our system at modest scale, we also built a coprocessor emulator. It simply runs the coprocessor's code on the worker's CPU, then waits an additional amount of time consistent with how much longer a coprocessor would have taken.

Finally, to evaluate our system at large scale, beyond the number of machines we have available, we built a simulator. It uses the measurements from our implementation to extrapolate the performance on thousands of machines. The simulator is 893 lines of Python.

7 Evaluation

In all experiments, unless stated otherwise, we use the following parameters: 2^{35} addresses; 10-KB blocks; 10,000 coprocessors; desired lifetime failure probability 2^{-80}; and ORAM lifetime 2^{50} requests. These are modeled after the map tiles data set. Because user interaction and fault tolerance are not yet implemented, we do not evaluate their performance impact.

When evaluating a storage system, it is typically vital to ensure the workload is realistic, e.g., by using traces of a real system. However, to avoid leaking information via timing channels, ORAM always has the same performance no matter what addresses are accessed and which accesses are reads or writes. Furthermore, since our algorithm never overlaps multiple accesses, workload burstiness also has no effect on performance. Therefore, workload is irrelevant and we use a simple synthetic one.

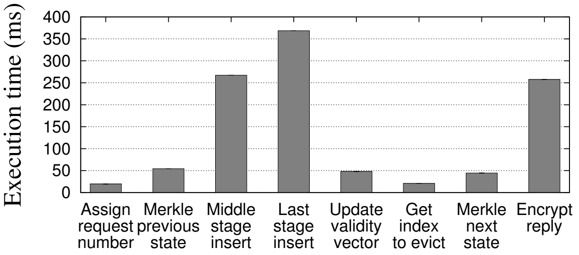


Figure 7: **Time to Perform Miscellaneous Algorithm Steps on the SLE 88.** *Times shown are for 1 KB block sizes. Error bars, generally too small to see, indicate 95% confidence intervals about the mean of 50 trials.*

7.1 Microbenchmarks

We evaluate how long different steps of our ORAM algorithm take on a real coprocessor, the Infineon SLE 88. We focus on three operations that consume significant amounts of time and must be repeated more than once, since these are the primary drivers of performance and opportunities for use of parallelism. These are *look up entry*, *evict from entry*, and *evict to entry*.

The left graph of Figure 6 shows the results of 50 trials for each step at each of five block sizes. We observe that, as expected, the dominant cost in all steps is proportional to block size.

The right graph of Figure 6 shows the predictions made by our simulator for these times. We see that they are in close agreement, and never off by more than 3%. Since we have few coprocessors, when we later perform simulations of data-center scale deployments we will use our validated simulator instead of real hardware.

For completeness, in Figure 7 we show the time taken by other algorithm steps implemented on the SLE 88.

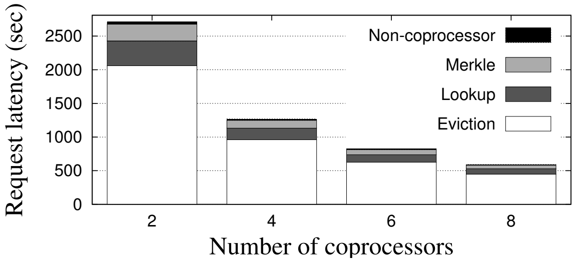


Figure 8: **Request Latency of Shroud.** *Total time per request for a 1 KB block, including defense against a fully malicious adversary. Non-coprocessor overhead is the part of average request time not accounted for by coprocessor run time.*

7.2 Benefits of Parallelism

To evaluate the parallelism Shroud can achieve, we employ a cluster of machines; each has 24 GB of RAM, two 10 Gbps NICs, and two quad-core Intel E5410, 2.33 GHz processors. As Figures 8 and 9 show, the secure coprocessors are the dominant bottleneck.

First, we measure parallelism on our full system. Due to limited budget, we only use eight coprocessors: Coprocessors are inexpensive when bought by the thousands, but expensive when bought individually. Figure 8 shows the results of these experiments; because of the modest scale, we use 1024-byte blocks and set $N = 2^{20}$, $M = 2^{20}$, and $\delta = 2^{-20}$ to make it finish in a reasonable amount of time. We see that by far the biggest component of operation time is time spent running on coprocessors, over 98% in all cases. Operation is highly parallelizable with this few coprocessors, so we see near-linear scaling with number of coprocessors.

Since the data set easily fits in memory, we do not incur disk latency. We expect that, under normal conditions, reading all the entries needed for request processing will incur only modest disk latency with careful disk scheduling, since it is amenable to bulk prefetching. The disk-scheduling element, however, is future work.

To evaluate on a larger scale, we use our coprocessor emulator in a cluster with 139 machines. We run one scheduler, 10 file servers, and up to 128 workers. Figure 9 shows the results of this experiment; again, we use 1024-byte blocks and set $N = 2^{20}$, $M = 2^{20}$, and $\delta = 2^{-20}$. For small numbers of coprocessors, the emulated results match their real counterparts to within 6%. We see, as before, that most of the time is spent by coprocessors, and that performance scales essentially linearly. This is not surprising since, thanks to the high parallelism of our design, there is nearly always parallel work available to keep all coprocessors busy.

To evaluate parallelism on a massive scale, we now use our simulator. We vary the number of coprocessors used for ORAM and measure the time to perform a request. Recall that the algorithm treats reads and writes identically, so it does not matter which we do.

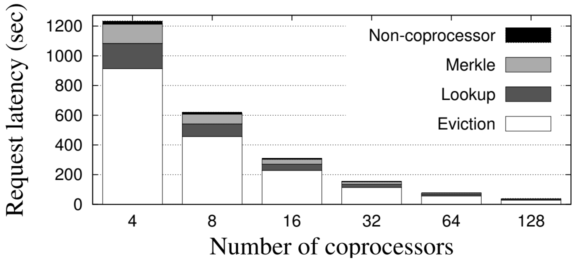


Figure 9: **Request Latency of Server with Emulated Coprocessors.** *Total time per request for a 1 KB block, including defense against a fully malicious adversary.*

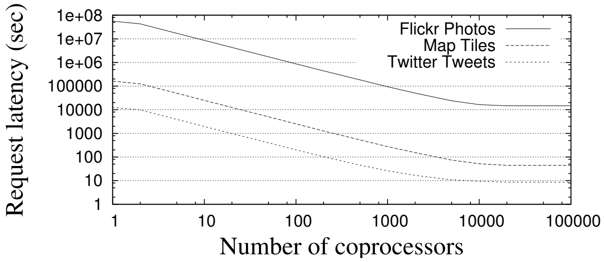


Figure 10: **Varying Number of Infineon SLE 88 Coprocessors.** *The effect of number of coprocessors on ORAM request latency for various workloads using simulated SLE 88 coprocessors. X-axis and Y-axis are log-scale.*

Figure 10 shows the effect of varying the number of coprocessors for various workloads using SLE cards. We see that parallelism is quite effective, reducing the time by over three orders of magnitude. For the Twitter tweets workload, it reduces the time from 13,000 sec to 8.6 sec, and for the map tiles workload, it reduces it from 160,000 sec to 44 sec. For the Flickr photos workload, it also reduces it substantially, from 16,000 hours to 4.1 hours, but this is still an unreasonably long period of time. This is due to the large block size, 5 MB; recall that our ideal performance is $O(RB)$ even without defense against malice.

Latency reduction is essentially linear in the number of coprocessors, since thanks to our restructuring, most of the expensive operations are highly parallelizable. A notable exception is the transition from one to two coprocessors, which reduces cost by much less than a factor of two because of the additional requirements for communication between the coprocessors. Also, effectiveness of parallelism stops after a certain number of coprocessors as we no longer have enough tasks to distribute among them. This occurs when we have about 19,000 coprocessors, enough to handle all entries in all buckets evicted to during the final stage.

Figure 11 shows the same experiment conducted with simulated IBM 4764 coprocessors. We see the same trend of highly effective parallelism, albeit with much lower latencies. However, since these cost $8,000 each, the low latencies shown are not practically attainable, at

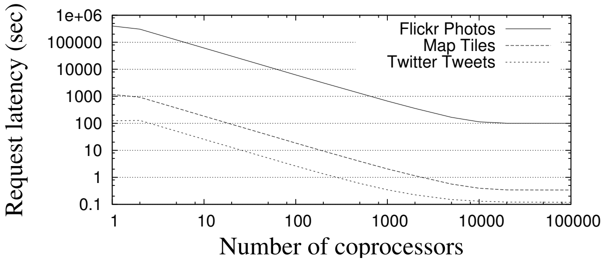


Figure 11: **Varying Number of IBM 4764 Coprocessors.** *The effect of number of coprocessors on ORAM request latency for various workloads using simulated IBM 4764 coprocessors. X-axis and Y-axis are log-scale.*

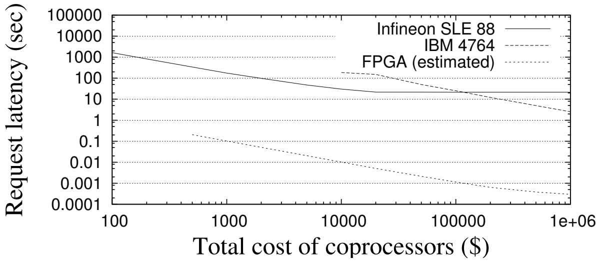


Figure 12: **Latency as a Function of Budget.** *Request latency as a function of money spent on coprocessors, for different coprocessor types. X-axis and Y-axis are log-scale. FPGA line assumes 1.4 Gb/s for all operations and $100 per FPGA.*

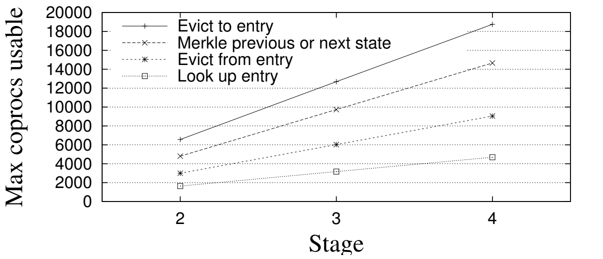


Figure 13: **Available Parallelism.** *How many coprocessors each step of the algorithm can make use of.*

least until technology trends make HSMs with this performance more affordable.

To fairly compare these two coprocessors, we conduct another experiment where we hold total coprocessor cost budget fixed and see how fast ORAM is with each coprocessor type. Figure 12 shows the results. We see that for low budgets, the inexpensive Infineon SLE cards offer better performance per unit cost. Only with more than about $100,000, when the SLE cards exhaust available parallelism, do the IBM 4764 coprocessors become suitable for use in our parallel ORAM. Finally, we estimate the performance of an FPGA-based implementation (§8).

To understand which steps of the algorithm are most amenable to parallelism, Figure 13 shows, for each parallelizable step, how many coprocessors it can make use of. We see that later stages can make use of more coprocessors, as they have larger ORAMs and thus more buckets. The step that can make most use of extra coprocessors is *evict to entry,* which updates all entries in evictee buckets. In contrast, *evict from entry* only touches entries of evictor buckets, of which there are half as many. The step that can make the least use of them is *look up entry,* which only updates entries of buckets along the designator path, of which there are half again as many.

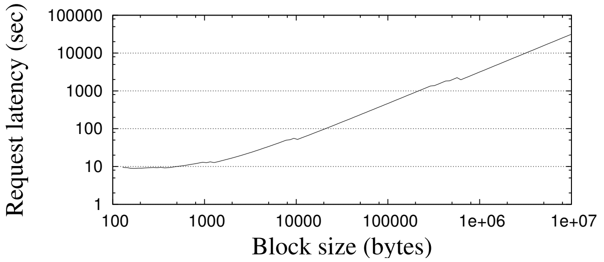


Figure 14: **Variable Block Size.** *Effect on parallel ORAM latency of varying block size. Simulation assumes* 2^{35} *blocks and 10,000 coprocessors. X-axis and Y-axis are log-scale.*

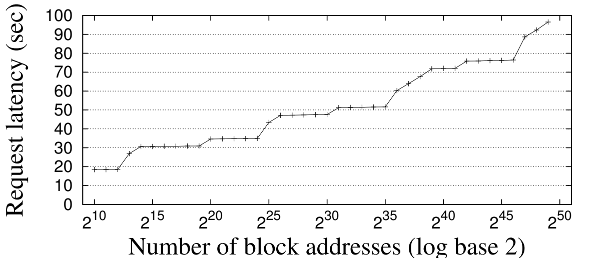


Figure 15: **Variable Block Count.** *Effect on parallel ORAM latency of varying number of addresses where blocks can be stored. Simulation assumes 10-KB blocks and 10,000 coprocessors. X-axis is log-scale and starts at* 2^{10} *blocks.*

7.3 Effect of Block Size and Count

In our next experiments, we look at the effect of varying workload characteristics on parallel ORAM performance. Specifically, we look at the effect of varying the size of blocks stored and the number of such blocks.

Figure 14 shows the effect of varying block size on ORAM performance, with all other parameters set to defaults. Below about 0.5 KB, request latency is under 10 sec, and below about 11 KB, it is under one minute. Thus, while ORAM may be feasible for workloads like Twitter tweets, Facebook images, and map tiles, it is unreasonable for the Flickr workload's 5-MB block size.

Another thing to observe is that at low block sizes, below about 1 KB, the effect of constant overheads dominates, and block size does not have much effect. After about 1 KB, each additional KB adds about 4.9 sec; after about 10 KB, each additional KB adds about 3.6 sec; and, after about 600 KB, each additional KB adds about 3.2 sec. The overhead drops at these points because larger block sizes allow more designators to be packed into a block, eventually reaching a tipping point where the number of stages R drops. Recall that our algorithm is $O(RB + R \log N)$.

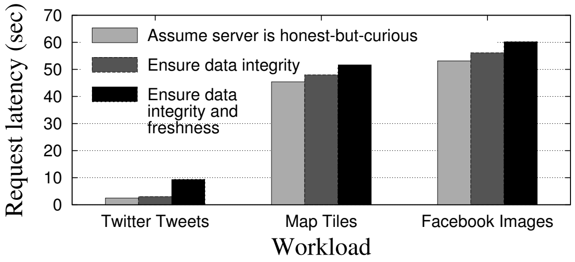


Figure 16: **Cost of Defense Against Malicious Servers.** *How ORAM latency increases due to defenses against malice.*

Figure 15 shows the effect of varying number of addresses on ORAM performance. We see that, unlike block size, address count has a less dramatic effect on request latency: in most cases, doubling it has no effect on performance. For some doublings, there is a large jump, due to the number of stages increasing by one; for other doublings, there is a small increase as the number of coprocessors required for a step goes above the next multiple of 10,000. For the range of block counts seen in our data sets, 2^{32} to 2^{36}, request latency is between 51 and 60 sec, but even data sets containing substantially more blocks than this would have comparable latency.

7.4 Cost of Defense Against Malice

Next, we evaluate the cost of defending against malicious servers by running experiments in which we turn off integrity and freshness checks. Figure 16 shows the results. The cost of these defenses is noticeable, demonstrating the importance of making their performance reasonable via parallelism and Merkle trees. For the Twitter tweets workload, the small block size means that the cost of communication dominates, and thus integrity and freshness checks are significant. Time increases by 21% for integrity, due to the need to compute and transmit MACs; adding freshness increases it by a further 278%, primarily due to Merkle tree maintenance. For the other workloads, where block sizes dominate hash sizes, integrity checks increase latency by only 6% and freshness checks by only 13–14%.

7.5 Comparison to the Hierarchical Scheme

To emphasize the importance of worst-case performance analysis, we simulate the traditional hierarchical algorithm [13, 14], which has $O(\log^3 N)$ amortized latency, but $\Omega(N)$ worst-case performance. Since the scheme is not parallelizable, for fairness we assume it runs on the fast IBM 4764. With 2^{35} blocks of 10 KB each, the average response time is 127 seconds. But, much more troubling are the slow responses. Every 10,000 requests, a request takes almost 92 minutes to complete, and every 10,000,000 requests, a request takes a week to complete! Since the entire service must wait for these requests to complete before serving the next request, this scheme is unsuited to the data center.

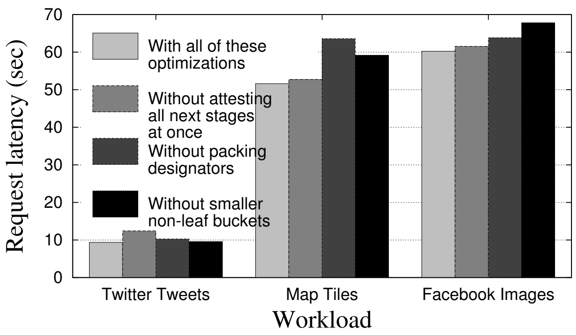


Figure 17: **Effect of Default Optimizations.** *Increase in ORAM latency when various optimizations we devised are removed. Simulation assumes 10,000 coprocessors.*

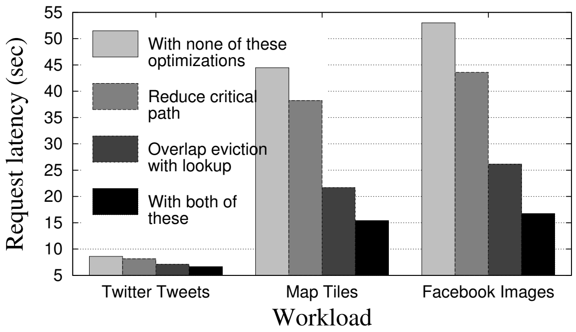


Figure 18: **Effect of Coprocessor-Intensive Optimizations.** *Decrease in ORAM latency when various additional optimizations we devised, which involve partially overlapping several stages, are added. Simulation assumes unlimited coprocessors.*

7.6 Effectiveness of Optimizations

We use our simulator to measure how effective certain optimizations are. We start with the standard optimizations we always apply: attesting all next stages at once, packing designators, and using smaller non-leaf buckets. Figure 17 shows the results. We see that each optimization always helps, but their effect depends on the workload. Attesting all next stages at once is highly useful for the Twitter tweets workload, reducing its latency by 25%. This is because this workload has a small block size, so attestation is a sizeable fraction of total run time; for the other workloads, the effect is small. Packing designators is highly useful for the map tiles workload, reducing its latency by 19%. The large effect is because it causes the number of stages R to drop from five to four; recall that our algorithm's performance is $O(RB)$. Smaller non-leaf buckets have a large effect on the map tiles and Facebook images, reducing latency by 13% and 11% respectively; although non-leaf buckets account for only half the buckets, they account for nearly all of the buckets accessed during the algorithm.

We also described optimizations that involve leveraging many extra coprocessors to overlap steps of several stages at once. One is to reduce the critical path by outputting designators early in the lookup process, and the other is to overlap eviction with lookup. To evaluate the effectiveness of these optimizations, we performed simulations with an unlimited number of coprocessors.

The results are in Figure 18. We see that the effectiveness of these optimizations is substantial for the workloads with large block sizes, and also significant for the small-block Twitter tweets. For Twitter tweets, reducing the critical path reduces latency by 5%; overlapping eviction with lookup decreases it by 17%; and doing both reduces it by 23%. For the others, reducing the critical path reduces latency by 14–18%; overlapping eviction with lookup decreases it by 51%; and doing both reduces it by 65–68%.

These large effects indicate it pays off substantially to use these optimizations when many coprocessors are available. Recall that Figure 13 discussed how many coprocessors would be necessary to overlap all stages at once, as these optimizations demand. However, even without that many coprocessors, it may be possible with intelligent scheduling to make partial use of the optimizations during periods when not all coprocessors are in use. Devising such schedules is future work.

7.7 Summary

Using parallelism substantially reduces the cost of ORAM, e.g., from 13,000 sec to nine seconds in the case of Twitter tweets. However, the resulting latency is still substantially more than that of a non-oblivious access. Such an access would have latency primarily caused by WAN latency and SSL overhead, and so would be only tens to hundreds of milliseconds. Clearly, more work is needed to bring ORAM latencies to full practicality.

8 Future Work

There are many avenues for extending Shroud. Most importantly, since we found secure coprocessor bandwidth to be the limiting factor, it would be useful to explore higher-bandwidth coprocessors, such as secure FPGAs, which may be able to offer multiple Gb/s of bandwidth. To estimate the potential of this approach, the "FPGA" line in Figure 12 simulates ORAM performance using estimates of FPGA capabilities.

If we can overcome the coprocessor bandwidth bottleneck, an important next consideration will be server storage latency. One way to reduce it is to use solid-state drives. However, it may be possible to just use disks and schedule requests so as to reduce seek time. After all, most disk accesses are due to eviction, which can be anticipated well in advance. By providing these access requests to the disk early, we may enable bulk prefetching.

Another concern is to improve not only latency but throughput of ORAM operations. One technique, which takes advantage of our approach of using many coprocessors in parallel, is to batch requests together and look them up in parallel. Note, however, that if requests in a batch use the same address, we must replace all but one with a random address so an adversary does not learn of the match. Also, after a batch is complete, we must replace the result of each modified request with that of the corresponding unmodified one.

9 Conclusion

Securing user data within the data center requires more than encryption; hiding data access patterns adds additional protection against malicious employees and hackers. We showed how to achieve this with Shroud, a distributed storage system that hides all information about block accesses. We show that deploying such oblivious storage in the data center creates new challenges and opportunities, including issues of scale, parallelism, maliciousness, fault tolerance, and worst-case performance. Our evaluation shows the usefulness of our approach, particularly our use of many secure coprocessors in parallel, in scaling ORAM to data-center-scale workloads.

Acknowledgments

The authors thank Sumeet Bajaj and Radu Sion for providing information about the IBM 4764, Barry Bond and Shrinath Eswarahally for helping us with the Infineon SLE 88 cards, and Jeremy Elson for assisting us with the FDS cluster. We are especially grateful to our shepherd, Ethan Miller, and the anonymous reviewers for their helpful comments.

References

[1] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring private access to large-scale data in the data center," in *Proceedings of the USENIX Symposium on File and Storage Technologies (FAST)*, Feb. 2013.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[3] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebooks photo storage," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2010.

[4] J. Guynn, "Google fires employee for snooping on users," *Los Angeles Times*, Sept. 16, 2010.

[5] W. Andrews, "China leadership 'orchestrated Google hacking'." *BBC News*. http://www.bbc.co.uk/news/world-asia-pacific-11920616, Dec. 4, 2010.

[6] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: An online social network with user-defined privacy," in *Proceedings of ACM SIGCOMM*, Aug. 2009.

[7] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[8] C. Troncoso, C. Diaz, O. Dunkelman, and B. Preneel, "Traffic analysis attacks on a continuously-observable steganographic file," in *In the International Information Hiding Workshop*, 2007.

[9] R. Kumar, J. Novak, B. Pang, and A. Tomkins., "On anonymizing query logs via token-based hashing," in *Proceedings of the World Wide Web Conference (WWW)*, 2007.

[10] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proc. of the USENIX Security Symposium*, Aug. 2004.

[11] E. Lee and C. Thekkath, "Petal: Distributed virtual disks," *SIGOPS Operating Systems Review*, vol. 30, Sept. 1996.

[12] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1987.

[13] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1990.

[14] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM (JACM)*, vol. 43, no. 3, 1996.

[15] E. Shi, H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *Proceedings of AsiaCrypt*, Dec. 2011.

[16] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[17] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proceedings of CRYPTO*, 2010.

[18] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *the Network and Distributed System Security Symposium*, 2012.

[19] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious RAM simulation with efficient worst-case access overhead," in *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, Oct. 2011.

[20] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012.

[21] F. Olumofin, *Practical Private Information Retrieval*. PhD thesis, University of Waterloo, 2011.

[22] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," in *Proceedings of VLDB*, Aug. 2008.

[23] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell, "Flat Datacenter Storage," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2012.

[24] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," 2009. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[25] Amazon, "Amazon Simple Storage Service," 2012. http://aws.amazon.com/s3/.

[26] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of FAST*, Jan. 2002.

[27] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, Oct. 1995.

[28] R. Sion and B. Carbunar, "On the practicality of private information retrieval," in *Network and Distributed System Security Symposium*, 2007.

[29] C. Aguilar-Melchor and P. Gaborit, "A lattice-based computationally-efficient private information retrieval protocol," in *the Western European Workshop on Research in Cryptology (WEWoRC)*, 2007.

[30] F. Olumofin and I. Goldberg, "Revisiting the computational practicality of private information retrieval," in *Proceedings of the Financial Cryptography and Data Security Conference*, Feb. 2011.

[31] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *Proceedings of the ACM Symposium on Theory of Computing*, 1997.

[32] A. Iliev and S. Smith, "Protecting user privacy via trusted computing at the server," *IEEE Security and Privacy*, vol. 3, no. 2, 2005.

[33] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," *IBM Systems Journal*, vol. 40, no. 3, 2001.

[34] D. Asonov and J.-C. Freytag, "Almost optimal private information retrieval," in *the Privacy Enhancing Technologies Symposium*, 2003.

[35] P. Williams and R. Sion, "Usable PIR," in *the Network and Distributed System Security Symposium (NDSS)*, 2008.

[36] P. Williams and R. Sion, "PrivateFS: A parallel oblivious file system," in *the ACM Conference on Computer and Communications Security*, 2012.

[37] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *ACM Conference on Computer and Communications Security*, 2012.

[38] M. Backes, A. Kate, M. Maffei, and K. Pecina, "ObliviAd: Provably secure and practical online behavioral advertising," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

[39] S. W. Smith, "Outbound authentication for programmable secure coprocessors," *Journal of Information Security*, vol. 3, 2004.

[40] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh, "Location privacy via private proximity testing," in *the Network and Distributed System Security Symposium (NDSS)*, 2011.

[41] Twitter, "#numbers." `http://blog.twitter.com/2011/03/numbers.html`, Mar. 2011.

[42] Twitter, "Measuring tweets." `http://blog.twitter.com/2010/02/measuring-tweets.html`, Feb. 2010.

[43] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2010.

[44] Flickr, "6,000,000,000." `http://blog.flickr.net/en/2011/08/04/6000000000/`, Aug. 2011.

[45] Infineon Technologies, "Security & chip card ICs, SLE 88CFX4000P." `http://www.datasheetcatalog.org/datasheets/228/339421_DS.pdf`, 2006.

[46] IBM, "CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 PCI-X cryptographic coprocessors." 19th Ed., 2008.

[47] K. Goldman, R. Perez, and R. Sailer, "Linking remote attestation to secure tunnel endpoints," Tech. Rep. RC23982, IBM, 2006.

[48] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2008.

[49] R. Canetti and H. Krawczyk, "Analysis of key-exchange protocols and their use for building secure channels," in *Proceedings of EuroCrypt*, 2001.

[50] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of CRYPTO*, 1987.

[51] D. A. Wheeler, "SLOCCount." `http://www.dwheeler.com/sloccount/`.

[52] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and secure message authentication," in *Proceedings of CRYPTO*, 1999.

A Proof of Correctness and Security of Oblivious Aggregation

Correctness. The protocol from §5.2.7 produces the correct output (namely $\bigoplus_{m=1}^{q} v_m$) by the following reasoning.

$$Z = \bigoplus_{m=1}^{q} z_m = \bigoplus_{m=1}^{q} x_m \oplus v_m = \bigoplus_{m=1}^{q} x_m \oplus \bigoplus_{m=1}^{q} v_m$$

The x_m are computed such that $\bigoplus_{m=1}^{q} x_m = 0$, so $Z = \bigoplus_{m=1}^{q} v_m$.

Security. We prove Theorem 1, showing that the oblivious aggregation protocol from §5.2.7 leaks nothing about the individual values v_m to the server. Such a leak would allow us to break the PRF.

Theorem 1 *Let PRF be a pseudorandom function. Let* π *be the functionality where each of* q *parties (coprocessors) has input* v_m *for* $1 \leq m \leq q$*, and another party (server) receives as output* $S = \bigoplus_{m=1}^{q} v_m$*. Then the oblivious aggregation protocol is a multiparty computation protocol for* π*, which provides security in the presence of a semi-honest server.*

Proof of Theorem 1: Our proof will be in the real/ideal world model. We define the following ideal functionality. It receives from each secure coprocessor a value v_m and outputs the sum $S \leftarrow \bigoplus_{m=1}^{q} v_m$. The simulator needs to simulate the interaction between the secure coprocessors and the untrusted server. To do so, the simulator obtains the sum S from the ideal functionality, computes q additive shares of S, and sends the shares to the server. The random shares $\{y_m\}_{m=1}^{q}$ are computed as follows $y_m \overset{R}{\leftarrow} \mathcal{V}$ if $m < q$, and $y_q = S \oplus \bigoplus_{m=1}^{q-1} y_m$.

Now we need to show that an adversary controlling the server cannot distinguish whether he is given $\{y_m\}_{m=1}^{q}$ from the simulator or $\{z_m\}_{m=1}^{q}$ from the secure coprocessors in the real execution. As an intermediate step, we consider the distribution $\{w_m \leftarrow v_m \oplus r_m\}_{m=1}^{q}$, where each r_m for $m < q$ is chosen uniformly at random and $r_q = \bigoplus_{i=1}^{q-1} r_i$. We now prove indistinguishability in two steps: first, we show that $\{y_m\}_{m=1}^{q}$ is indistinguishable from $\{w_m\}_{m=1}^{q}$, and then that $\{w_m\}_{m=1}^{q}$ is indistinguishable from the distribution $\{z_m\}_{m=1}^{q}$ in the real execution. The first point follows from the fact that r_m for $m < q$ are chosen uniformly at random and hence the same is true for $w_m = v_m \oplus r_m$ when $m < q$, and $w_q = S \oplus \bigoplus_{m=1}^{q-1} w_m$. Therefore the values $\{w_m\}_{m=1}^{q}$ have the same distribution as $\{y_m\}_{m=1}^{q}$.

We now reduce the indistinguishability of $\{w_m\}_{m=1}^{q}$ and $\{z_m\}_{m=1}^{q}$ to the indistinguishability of the output of a PRF from a truly random function. Assume there exists an adversary $\mathcal{A}$ who distinguishes these two distributions with non-negligible probability; we use this adversary to construct an adversary $\mathcal{A}_{PRF}$ who distinguishes the output of the PRF from random. The adversary $\mathcal{A}_{PRF}$ is given access to an oracle $O_F(\cdot)$, and his goal is to guess whether this oracle implements a PRF F_K or a truly random function F. We construct $\mathcal{A}_{PRF}$ as follows:

1. $\mathcal{A}_{PRF}$ picks q values at random: $v_m \overset{R}{\leftarrow} \mathcal{V}$.
2. $\mathcal{A}_{PRF}$ queries his oracle $O_F(\cdot)$ in order to compute the values x_m as follows: $x_m = O_F(j||i) \oplus O_F(j||(i+1 \bmod q))$ for $1 \leq m \leq q$.
3. $\mathcal{A}_{PRF}$ sends $\{v_m \oplus x_m\}_{m=1}^{q}$ to $\mathcal{A}$.
4. If $\mathcal{A}$ guesses that he was given $\{w_m\}_{m=1}^{q}$, then $\mathcal{A}_{PRF}$ guesses that he was interacting with a random function. Otherwise, $\mathcal{A}_{PRF}$ guesses that he was interacting with a pseudorandom function.

We observe that if $O_F(\cdot)$ is implemented with a PRF, then the input provided to $\mathcal{A}$ is of the same form as $\{z_m\}_{m=1}^{q}$. On the other hand, if $O_F(\cdot)$ returns answers from a truly random function, then the values that $\mathcal{A}$ receives are of the same form as $\{w_m\}_{m=1}^{q}$. Therefore, $\mathcal{A}_{PRF}$ guesses correctly and distinguishes the output of the PRF from random with non-negligible, contradicting the security of the PRF. ■

B Proof of Correctness of Delegation

In §5.4, we described a protocol for delegating authority for request number assignment to a new set of master instances. Here, we prove that this is safe, i.e, that it cannot cause two different requests to be assigned the same request number.

Theorem 2 *No two distinct requests are assigned the same request number.*

Proof of Theorem 2: Before beginning, we make the following observation. A master set can be attested in only two ways: it can be the unique master set attested at service creation time, or it can be attested via delegation. A master set attested via delegation is derived from an earlier master set attestation, along with a quorum of delegation attestations. Thus, each master set attestation must be the final link of a chain of attestations, starting with the unique original master set attestation and continuing via a sequence of delegations.

Now, suppose the protocol is not safe, and two requests R and R' are assigned the same number r. For this, the server must have gotten one master set quorum to assign r to R and another to assign r to R'. Since no single master instance can ever assign the same request number twice, the quorums must not overlap. Since quorums of the same master set overlap, the quorums must be from two different master sets. Thus, there exist two different master sets $\mathcal{S}_1$ and $\mathcal{S}_2$, both having assigned the same request number r.

Consider the chains for $\mathcal{S}_1$ and $\mathcal{S}_2$. They share at least one common element, since they both start at the same original attestation. Thus, they have a last common element, which we'll call the latest common predecessor. There are two cases to consider: (1) this common predecessor is one of $\mathcal{S}_1$ and $\mathcal{S}_2$, or (2) the common predecessor is a different master set $\mathcal{S}'$.

Consider case 1, and without loss of generality assume the common predecessor is $\mathcal{S}_1$. We thus know that $\mathcal{S}_1$ delegated responsibility, starting with some request number x, to a subsequent master set, and this was the beginning of a chain of delegations ending in $\mathcal{S}_2$. Since we know $\mathcal{S}_1$ assigned request number r, and it cannot delegate a request number it has already assigned, we know $x > r$. However, this means that $\mathcal{S}_2$ was never delegated responsibility for requests $< x$, and in particular was never delegated responsibility for r. This contradicts the assumption that it assigned a request to r.

Now, consider case 2. In this case, we have a master set $\mathcal{S}'$ that is a latest common predecessor of two distinct master sets $\mathcal{S}_1$ and $\mathcal{S}_2$. Thus, it must have delegated responsibility to two distinct master sets, its successor in the chain leading to $\mathcal{S}_1$ and its successor in the chain leading to $\mathcal{S}_2$. However, a master set cannot delegate to another without a quorum of its members delegating. Quorums overlap, so there must be at least one member of $\mathcal{S}'$ that delegated to both successors. This is impossible, since a master instance destroys itself upon delegation, and therefore cannot delegate a second time.

We have covered all cases and shown them to be impossible, so the original supposition must be false and the delegation protocol is safe. ■

C Proof of Safety for Reduced Entry Count

In §5.5, we gave a brief explanation of why non-leaf buckets require fewer entries than non-leaves. We now present a full proof.

Theorem 3 *With only* $\log_2(M(D-2)(R-1)/\delta)$ *entries per non-leaf bucket, where D is the depth of the tree the bucket is in, the probability of overflow, in the course of M operations, is* $\leq \delta$.

Proof of Theorem 3: Let $\mathsf{Ov}(E)$ be the event that, within the first M ORAM operations, at least one non-leaf bucket in the tree exceeds size E. Let $\mathsf{Ov}'(E)$ be the event that, within the first M ORAM operations, at least one non-leaf bucket goes from size E to $E+1$ as a result of its parent performing an eviction. It is clear that $\mathsf{Ov}'(E) \Rightarrow \mathsf{Ov}(E)$, since if at least one non-leaf bucket exceeds size E after its parent performs an eviction, then it exceeds size E. We can also show that $\mathsf{Ov}(E) \Rightarrow \mathsf{Ov}'(E)$, as follows. If a non-leaf bucket exceeds size E within the first M ORAM operations, then there must be a first time that it exceeds size E during those ORAM operations. This must happen right after its parent performs an eviction, since there is no other way for it to go from not exceeding E to exceeding E. At this point, it goes from size E to $E+1$.

Since $\mathsf{Ov}'(E) \Rightarrow \mathsf{Ov}(E)$ and $\mathsf{Ov}(E) \Rightarrow \mathsf{Ov}'(E)$, we can conclude $P(\mathsf{Ov}(E)) = P(\mathsf{Ov}'(E))$.

The root and depth-1 buckets can never overflow because they are evicted after every operation. Thus, there are only $D-2$ levels containing non-leaf buckets that can overflow. Every operation, four buckets from each of these levels has a parent that may perform an eviction to it. Thus, after every operation, there are at most $4(D-2)$ non-leaf buckets that may go from size E to $E+1$ as a result of an eviction from its parent. Altogether, then, there are $4M(D-2)$ opportunities for a non-leaf bucket to do this.

For any $1 \leq i \leq 4M(D-2)$, let $\mathsf{Ov}'(i,E)$ be the event that, during the ith opportunity for a non-leaf bucket to overflow due to an eviction from its parent, it actually does so. Consider one of these opportunities, in which a bucket b has a parent selected for eviction. The probability of b actually going from size E to $E+1$ in this way can be bounded as follows. For this to happen, b's parent must have positive load, the entry evicted from b's parent must have a designator that points toward b, and the load of b must be exactly E. These are all independent events, so their probabilities can be multiplied. The probability that b's parent has positive load is $\leq 1/2$. The probability that the entry b's parent evicts has a designator that points toward b is $1/2$. The probability that b's load is E is $\leq$ the probability that b's load is at least E, which is $\leq 1/2^E$. Altogether, we have the probability that b goes from E to $E+1$ is $\leq 1/2^{(E+2)}$. In other words, $P(\mathsf{Ov}'(i,E)) \leq 1/2^{(E+2)}$.

Since these are the only opportunities for there to be overflow in this way, we know $\mathsf{Ov}'(E) = \bigcup_{i=1}^{4M(D-2)} \mathsf{Ov}'(i,E)$. By the union bound, we thus have $P(\mathsf{Ov}'(E)) \leq \sum_{i=1}^{4M(D-2)} P(\mathsf{Ov}'(i,E))$. Since we've shown that $P(\mathsf{Ov}'(i,E)) \leq 1/2^{(E+2)}$, we can conclude that $P(\mathsf{Ov}'(E)) \leq \frac{4M(D-2)}{2^{(E+2)}} = \frac{M(D-2)}{2^E}$. Since we showed that $P(\mathsf{Ov}(E)) = P(\mathsf{Ov}'(E))$, we have $P(\mathsf{Ov}(E)) \leq \frac{M(D-2)}{2^E}$.

To ensure that the probability of overflow is at most δ, we want to make the probability of overflow in any tree be at most $\delta/(R-1)$, since there are $R-1$ trees with buckets that can overflow. Thus, we use a number E of entries per bucket such that $P(\mathsf{Ov}(E)) \leq \delta/(R-1)$. This holds if $\frac{M(D-2)}{2^E} \leq \delta/(R-1)$, which holds if $E \geq \log_2[M(D-2)(R-1)/\delta]$. ■

D Proof of Security of the Parallel ORAM Constructions

Theorem 4 *The parallel ORAM construction from Section 5.2 provides query access-pattern hiding in the presence of a semi-honest server.*

Proof Sketch: In order to show the access-pattern hiding property of the scheme, we construct a simulator S that outputs a simulated view of the server for the execution of a sequence of q queries that is indistinguishable from the server's view in the real execution of the queries.

Since the underlying Binary Tree ORAM construction is secure, we know that there is a simulator that outputs a sequence of physical locations that is indistinguishable from the physical addresses accessed by the server in the real execution of the protocol. S uses this simulator to simulate the nodes requests by the co-processors at each ORAM access.

In order to simulate the values that the server receives from the k co-processors in order to compute the final output returned to the client, S computes k random shares of a random encryption, $Enc(0)$, of 0.

The above simulated view is indistinguishable from the server's view in the real execution for the following reasons. The simulated physical locations are indistinguishable from the real accessed locations because of the properties of the underlying tree ORAM simulator. The sequence of real and simulated shares are also indistinguishable, which we prove with the following hybrid argument:

- The simulated view is indistinguishable from a real execution in which the searched value is $Enc(0)$. This follows from the security of the oblivious aggregation in Theorem 1.
- The real execution when the searched value is $Enc(0)$ is indistinguishable form the real execution with the actual search value $Enc(data)$. This follows from the semantic security properties of the encryption scheme used for the stored database.

The simulator chooses buckets for eviction according to the prescribed probabilities in the protocol. The correctness of eviction is guaranteed by the fact that the server is semi honest.

Theorem 5 *The parallel ORAM construction obtained after the modifications from Section 5.3.1 and Section 5.3.2 provide query access pattern hiding in the presence of a malicious server.*

Proof Sketch: We construct a simulator for the server's view in the same way as in the proof of Theorem 4. However, we would need to use the scheme modifications in order to claim the indistinguishability in the two steps of the hybrid argument as follows.

- Our argument that the random shares of $Enc(0)$ are indistinguishable from the shares produced by the co-processors for each query relied on the fact that the output of a PRF evaluation with the same key and the same input is used only once, which required that for each query the co-processors use different keys (or different seeds) for their PRF evaluations. However, since the co-processors are not interacting during a query execution, they need to be able to derive a different key for each query from their input for the query execution. The unique query number identifier maintained by the master co-processor provides this property.
- The argument that the co-processor shares of $Enc(0)$ are indistinguishable from the shares of the co-processors produced during the real query execution relied on the fact that the shared output of the co-processors is never the same value across two queries. This property is guaranteed by the properties of the underlying tree ORAM scheme as long as the co-processors are given as input the correct state of the database after the execution of the preceding queries. The Merkle tree maintained over the database guarantees that the co-processor always receives the correct state of the ORAM memory storage.

Finally the simulation of the physical addresses that the server accesses in the database is indistinguishable from the addresses accessed during the real execution if and only if the real execution for each query uses as input the correctly updated ORAM storage. Again this holds because of the Merkle tree properties.

The fact that the co-processors select the buckets for eviction guarantees that the scheme will maintain the right eviction probabilities for the different buckets; i.e., the server will not be able to bias the buckets chosen for eviction in such a way that some buckets will never be evicted and will overflow with higher probability.

Thus, we conclude that the simulated server's view is indistinguishable from his view in the real execution, even when the server is malicious.