Toward Practical Private Access to Data Centers via Parallel ORAM

Jacob R. Lorch, James Mickens, Bryan Parno — *Microsoft Research*
Mariana Raykova — *Columbia University*
Joshua Schiffman — *Pennsylvania State University*

Abstract

Today, accessing maps, pictures, status updates, and other data from online services is de rigueur, but these accesses may leak private information. Previous work proposed using a secure coprocessor at the server to hide all information about user requests via an oblivious RAM (ORAM) protocol. For this to ever be practical, ORAM must be adapted to the exigencies of the data center. We explore the changes needed for such adaptation. We show, via new techniques for oblivious aggregation, how to securely use many secure coprocessors acting in parallel to improve request latency. Despite the challenges of the distributed setting, we protect against fully malicious servers and coprocessor faults. Our evaluation combines large-scale simulations with an implementation on a secure coprocessor and suggests that these adaptations bring ORAM in the data center closer to practicality.

1 Introduction

Requesting data from an online service reveals potentially sensitive information about the request and response. Using SSL hides this information from eavesdroppers, but not from the service itself. Even if the data is stored encrypted, the pattern of data accesses may reveal information. A series of map tile retrievals may reveal your commuting habits, retrieving tweets from Twitter may reveal your interests, and retrieving photos from Facebook may reveal your social graph.

In this work, we investigate techniques to provide privacy-preserving access to data-center-scale online services. These services typically host tens of billions of records (§4.1); traditional protocols for making the server oblivious to access patterns simply do not scale to this much data (§2.1). Instead, we turn to schemes based on secure hardware [16]. In these schemes, the user creates a cryptographically-secure channel to a secure, trusted coprocessor running on the server, and uses this channel to submit requests and receive replies (see Figure 1). The coprocessor performs requested reads and writes by using an efficient oblivious RAM (ORAM) scheme [12, 13, 23]; thus, the server learns nothing. Relative to the cost of a non-private data access, the user pays only the CPU and bandwidth overhead of an SSL connection. Only the server pays the cost, within the data center where bandwidth is plentiful and cheap.

However, employing ORAM in a data center raises numerous issues not previously considered in the ORAM literature (§2). What is different about ORAM in the data center? First, the **size** of the data sets (§4.1) and the **limited resources** of secure coprocessors (§4.2) make many schemes impractical. For example, schemes that require the coprocessor to store $O(\sqrt{N})$ records [30, 34] would exceed the memory available on modern secure coprocessors by three to six orders of magnitude.

Second, modern data centers are designed around **parallelism**, a notion not previously explored in the ORAM literature. Just as a modern web service uses thousands of computers in parallel, ORAM schemes for the data center should be designed to exploit thousands of secure coprocessors working together (§7). Indeed, secure hardware has become so commoditized that it would be quite reasonable to outfit most computers in a data center with their own secure coprocessors (§4.2). Exploiting this parallelism naïvely can be inefficient and/or insecure (§3.3). Parallelism also provides an **arbitrarily malicious** server with more opportunities for mischief, and standard mechanisms for coping with such mischief (like integrity protecting the database) make parallel operation difficult. Nonetheless, we enhance our parallel ORAM scheme to cope with arbitrarily malicious servers (§5.3), with little impact on the parallelism of the scheme.

Next, using thousands of processors in a data center implies regular hardware failures, making **fault tolerance** critical to algorithm design. §5.4 shows how to deal securely and efficiently with coprocessor failures.

Finally, most ORAM schemes analyze performance in an amortized model [12, 13, 16, 23, 25, 34, 35], whereas **worst-case** guarantees matter to online services [2, 36]. Since they depend on 24x7 operation to keep customers happy, they cannot afford long downtimes. However, with most ORAM schemes, the bad performance hidden by the amortized analysis occasionally stops the entire service for long periods (e.g., 92 hours to a week – §7.5), during which no requests can be handled. A few efforts have considered worst-case ORAM performance [14, 18, 24, 26], and we applaud these efforts as better suited to data center expectations. To emphasize this point, we concretely demonstrate the deleterious effects of amortized analysis in §7.5.

We implement the basic functionality of our scheme on Infineon SLE 88 smart cards (§6), and use this implementation to evaluate the performance of our protocols (§7). We find that parallelism can bring response times down to seconds, for many real world data sets, even when protecting against malicious servers. Data sets with large records, however, are costly to protect. Achieving this parallelism requires the purchase of thousands of coprocessors, but at $4 each, a data center should be able to afford this outlay.

2 Background and Related Work

We provide a brief overview of work on PIR and ORAM. For more details see [21] or [18].

2.1 Private Information Retrieval (PIR)

Traditional PIR. In a private information retrieval (PIR) protocol [9], a user retrieves record i from a database of N items held by a server, without the server learning i. Sion et al. [27] 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 [1], can outperform the trivial PIR scheme [22]. Nonetheless, Olumofin and Goldberg's results [22] indicate that even these schemes require over 25 minutes to access one record 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 (see §4.1).

Multi-Server PIR. Using multiple non-colluding servers is another way to improve on the efficiency of both information theoretic and computational solutions [9, 21, 22, 24]. Nonetheless, these response times are still far too slow for data center workloads, and more importantly, the data sets we consider (see §4.1) are large enough to make duplication prohibitive and are held by companies that have 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 record and return it to the user via the secure channel (see Figure 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 utilize the same service.

Iliev and Smith first proposed this elegant combination of trusted hardware and ORAM for PIR [16], though the notion is implicit in the earlier work of Smith and Safford [29] and Asonov and Freytag [3]. The former essentially uses the naïve ORAM scheme that reads all of the records for each request, while the latter proposes an algorithm quite similar to the classic "Square-Root" algorithm [12, 13].

2.2 Oblivious RAM (ORAM)

Classic. Goldreich and Ostrovsky [12, 13, 23] 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, 23]. These schemes include several serial steps, making it challenging to leverage a large number of distributed secure coprocessors. For example, they must read from cache i to decide which record to read from cache $i+1$. Furthermore, these schemes require periodic oblivious reshuffling of the data records, which introduces a serialization point.

Modern. A recent surge of interest in ORAM has explored various optimizations to the classic Hierarchical Scheme [13, 23], including the use of cuckoo hashing [14, 18, 25] and Bloom filters [35]; used incautiously, these optimizations can leak information [18]. 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 [18].

Traditional ORAM [13] 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 [30, 34]. Given the capabilities of modern secure coprocessors and the size of data center data sets (§4), these schemes are infeasible for our purposes. They would require gigabytes of coprocessor memory, when even high-end secure coprocessors have only 32 MB of memory, and the low-end have far less, e.g., 16 KB.

As we discuss in §1, most existing ORAM schemes measure performance in an amortized setting that is inappropriate for data center workloads [2, 36]. Exceptions include work by Ostrovsky and Shoup [24], who show how to deamortize the Hierarchical Scheme [13, 23] and achieve $O(\log^3 N)$ *worst-case* overhead. Recent work by Goodrich et al. [14] and Boneh et al. [7] also considers worst-case guarantees, but only at the cost of excessive (for the data center setting) storage on the coprocessor: $O(N^{1/\tau})$ and $O(\sqrt{N})$ respectively. Goodrich et al. [14] propose a second scheme with constant coprocessor storage, but the overhead is $O(\sqrt{N}\log^2 N)$. Finally, Kushilevitz et al. [18] show how to deamortize their $O(\log^2 N/\log\log N)$ scheme to provide the same worst-case guarantee with constant coprocessor storage.

For use in a datacenter environment, we selected the Binary Tree scheme of Shi et al. [26], which we describe in detail in §5.1. It is well-suited for our goals because it guarantees $O(\log^3 N)$ *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. Although the Binary Tree approach provides weaker security guarantees than previous schemes, the data center environment compensates for this weakness. Specifically, in most ORAM schemes, the adversary's probability of success decreases *exponentially* in the *security*

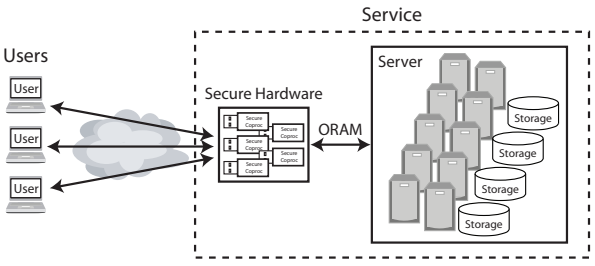

Figure 1: **Private Data Access Overview.** *To read and/or write data items, a user establishes a secure channel to a secure coprocessor running in the service's data center. Working with other coprocessors, the secure coprocessor uses an ORAM algorithm to retrive/update a database record without revealing to the service which record was accessed. The result is returned to the user over the secure channel.*

parameter for the scheme. In contrast, in the Binary Tree scheme, the adversary's probability of success decreases only *polynomially* in the size of the *data set*. Fortunately, for the data-center-scale data sets that we contemplate (see §4), the data sets are large enough to make this security guarantee sufficient–the adversary's probability of success against the ORAM will be no better than his probability of breaking the underlying cryptographic primitives (e.g., encryption and MAC schemes) directly. Shi et al.'s scheme also bounds the number of ORAM operations permitted; we remove this assumption (§5.5).

3 Problem Definition

3.1 Execution Model

Figure 1 provides an overview of our execution model. At a high level, remote users achieve private data access by indirecting their requests through secure coprocessors running inside the service's data center. Collectively, using the servers in the data center as untrusted storage, the secure coprocessors implement an ORAM algorithm, allowing them to service requests without leaking any 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. However, we assume he cannot violate the tamper-responding secure coprocessors. The attacker can also submit known queries to the service and observe the service's internal operations in response.

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 utilizing a collection of q secure coprocessors is replicated execution, i.e., 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 records. 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 q secure coprocessors share a key and operate on the same encrypted data set. Unfortunately, this may leak information about whether two requests are for the same address. Dealing with writes as well as reads adds additional complications.

As an example, in many ORAM schemes (e.g., [13, 18, 25]), the secure coprocessor maintains a cache of the most recently accessed items. With q secure coprocessors, one might imagine each secure coprocessor maintaining its own cache of d items, creating a collective cache of $q \cdot d$ items. Suppose these coprocessors operate independently on incoming requests, and imagine two requests arrive for item v, with each being 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 record, undermining the ORAM's intended obfuscation.

Typical secure coprocessors are I/O bound (§4.2), so

Data Set	# of Records	Record 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

Figure 2: **Datacenter-Scale Data Sets.** Approximate sizes.

coordinating coprocessors via explicit messages is too expensive. Instead, we require new protocols to synchronize the coprocessors and keep the ORAM secure.

4 Operating Constraints

To illustrate the formidable challenges of using ORAM in a data center, we describe the size of real data sets used by real web services, along with the computational limitations of current secure hardware.

4.1 Big Data

Modern web services expose vast data sets via public interfaces. Even if the data were encrypted, services could still glean sensitive information from user access patterns. Even visiting the service via an anonymous proxy [10] would not obviate this privacy threat.

We describe several representative data sets for which we have concrete numbers. Figure 2 summarizes them.

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 [20], retrieving map information may leak the location data anyway. 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 (zoom). The number and size of tiles varies by level, but the majority ($\sim 2^{35}$) are at level 19 with a maximum size of 10 KB, so we use this as our representative sample.

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. Thus, Twitter is a reasonable candidate for PIR. According to public reports [32], Twitter currently receives one billion tweets every week. Extrapolating from previously published metrics [31], 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 [4], 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 [5]. 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 [5], we choose this image size as a representative for the Facebook images.

Flickr Photos. Flickr is another popular photo sharing site with over six billion photos as of August 2011 [11]. Like Facebook, it stores versions of each photo in multiple sizes; unlike Facebook, it also makes the original image available. Typical consumer-grade cameras can produce images in the 3–14 MB range, so we somewhat arbitrarily choose 5 MB as a representative of a data set with larger records.

	Infineon SLE 88	IBM 4764
CPU	66 MHz	266 MHz
Memory	16 KB	32 MB
I/O	14 KB/s	9.85 MB/s
3DES 1 KB	75 KB/s	1.08 MB/s
SHA-1 1 KB	136 KB/s	1.42 MB/s
SHA-1 64 KB	145 KB/s	18.6 MB/s
SHA-1 1 MB	147 KB/s	22.5 MB/s
Cost	$4	$8,000

Figure 3: **Secure Coprocessor Performance** *Specs and measured performance of two representative coprocessors.*

4.2 Secure Hardware Performance

Today, there are a wide range of choices available for secure coprocessors. We selected a representative from each end of the spectrum and performed microbenchmarks to characterize their performance. These results are summarized in Figure 3. Both devices provide cryptographic accelerators, internal key storage, active protection against physical tampering, and the ability to attest to the code they run [28].

At the low end, the Infineon SLE 88 [17] is a small but surprisingly powerful and secure 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. While the SLE 88 is available in a variety of configurations, 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 [15] 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 crypto-

Symbol	Meaning
N	# of database records, each of size B
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

Figure 4: **Notation Summary.**

graphic 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. First, §5.1 presents the Binary Tree algorithm of Shi et al. [26]; the discussion introduces terminology that will aid our description, in §5.2, of how we modify it to add parallelism. §5.3 describes how to harden our new algorithm against malicious adversaries. §5.4 describes how to make our scheme tolerant to the failure of one or more secure coprocessors. §5.5 discusses additional optimizations.

5.1 The Binary Tree Algorithm

In this subsection, we describe the Binary Tree ORAM algorithm of Shi et al. [26], though we restructure the computations in a way that is useful for parallelism, fault tolerance, and dealing with a malicious server. Figure 5 provides a high-level overview, while Figure 18 provides detailed pseudocode.

5.1.1 Terminology

We assume that the data records stored in the ORAM have virtual addresses ranging from 1 to N. We further assume that all records have size B bits. We call anything of size B bits a *block*; we reserve the block value 0 to mean an invalid record. We assume there will be a maximum of M operations (in §5.5 we remove this assumption), and that we can tolerate a failure probability of δ during that sequence of operations. We assume $M = O(\text{poly}(N))$ and $1/\delta = O(\text{poly}(N))$. §A discusses additional parameter settings.

We support two kinds of user-initiated operations: read and write. Read accepts a virtual address and returns the record with that address. Write accepts a virtual address and new record, and overwrites the old record at that address. Either can accept a virtual address of 0, meaning that the operation is to be treated as a dummy operation with no change to the effective state. In practice, our scheme, like other ORAM schemes, treats reads and writes identically, meaning that the server cannot distinguish between them and that they have identical performance characteristics.

5.1.2 Overview

The ORAM construction of Shi et al. [26], uses a binary tree with N nodes called *buckets* (Figure 5). Each bucket contains $O(\log N)$ *entries*, each of which contains an address and record. 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 record 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, it is inserted into the root bucket of the binary tree. To prevent this bucket from overflowing, on each request, the coprocessor randomly selects two buckets from each level of the tree, for a total of $O(\log N)$ buckets, to *evict* from. When a bucket is selected, one of its valid entries is randomly removed and added to the child on the path down to the designator's leaf bucket.

When a user requests a read or write at an address, the coprocessor looks in its mapping for the address's designator. It then reads all of the buckets in the tree along the path between the root and this designator. When it finds the entry with the desired address, it removes that entry from its current bucket and eventually rewrites 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.

The table mapping addresses to designators must itself be accessed obliviously. However, it contains $O(N)$ mappings, making it far too large for the coprocessor to store. Fortunately, each mapping is tiny (e.g., 6 bytes), so many mapping entries fit in a single block. Thus, we can recursively apply the same ORAM scheme to this smaller collection of blocks. Each ORAM, or *stage*, will be progressively smaller, until we arrive at an ORAM that fits into 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, the main binary tree – stage R. R is generally quite small; indeed, for the data sets from §4.1, $R \approx 4$.

5.1.3 Storage

The server provides storage to support the ORAM service. Since the server is distrusted, coprocessors encrypt all sensitive data using SymmetricKey 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. The encrypted items are assigned a public identifier, so that the coprocessors can instruct the server which item to retrieve. These identifiers are chosen so that they re-

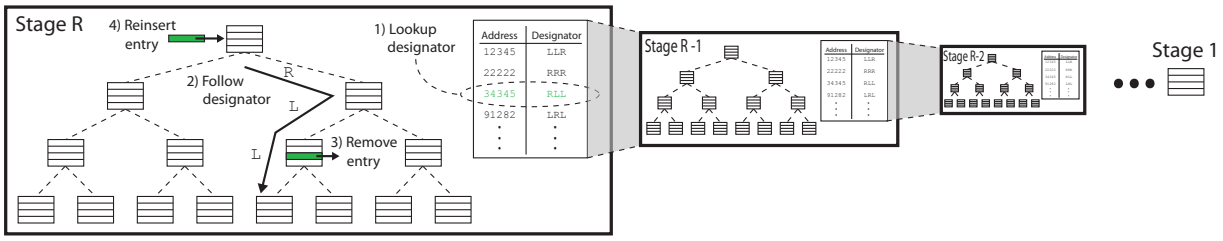

Figure 5: **Overview of the Binary Tree Algorithm.** *Each of the N buckets in the binary tree consists of $O(\log N)$ entries. To find the entry for a given address, we look up the address's designator, and follow the path through the tree dictated by that designator. For each bucket on the path, we 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, it is removed from wherever it was found, reinserted into the top-level bucket, and assigned a new designator. 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, so stage 1 fits within a single block.*

veal no information to the server.

5.2 Adding Parallelism

We discuss how to parallelize the Binary Tree scheme.

5.2.1 Initialization

To parallelize the algorithm, we need to run parts of it simultaneously on multiple coprocessors. Thus, we need a way to induct several coprocessors. 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 a fresh symmetric key SymmetricKey, an asymmetric key pair $\langle \text{PublicKey}, \text{PrivateKey} \rangle$, a key for a secure pseudorandom function PRF, and a seed UHFSeed for a universal hash function. It reveals N, B, UHFSeed, and PublicKey to the server, the user, and any future user who asks.

To join the service, C_{new} asks the server for PublicKey. Next, C attests to C_{new} that it is running the correct code in a secure environment [28], as well as to the validity of PublicKey. C_{new} then uses PublicKey to create a secure channel to C. Over this channel, C_{new} generates a similar attestation to convince C that it is a secure platform running the same code as C. Once satisfied, C provides C_{new} with the information needed to participate in the service, namely N, B, SymmetricKey, PrivateKey, UHFSeed, and PRF. Once C_{new} learns this information, it has joined the service.

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 [28]. 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. The user may then use 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. At the algorithm's termination, the coprocessor returns the result to the user over the secure channel.

5.2.3 Parallelism Goals

Our primary aim is to minimize the number of 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 $R = O(\log_c N)$ 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, it is four or less.

The primary bottleneck of the algorithm is the R entry lookups (i.e., invocations of `LookUpAndRemove` in Figure 18), each of which reads B-size entries from the server in two nested loops – one over the $D_s + 1$ buckets in the path and one over the E_s entries in each bucket. Since D_s and E_s are both $O(\log N)$, the overhead is $O_C(RB\log^2 N)$. This is substantial given how slowly coprocessors can receive, transmit, and process data.

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 reasonable latency (§7).

5.2.4 Parallelizing Bucket Accesses

We describe how to access entries in a bucket (essentially the helper functions in Figure 19) in $O_C(B)$ time.

Parallelizing entry updates (`Obliviously-UpdateEntry`) 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 (`ObliviouslyReadEntry`). 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 propose separating 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 $E_s = O(\log N)$ entries in all $D_s + 1 = O(\log N)$ buckets on the path indicated by the designator. Executed serially, this results in $O_C(RB\log^2(N))$ overhead. If instead we access all entries in parallel on $D_sE_s = O_C(\log^2(N))$ 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 [26]. 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`). Naively, 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 $2D_s - 1 = O(\log N)$ buckets and to twice that many, this takes $O_C(\log N)$ time.

Fortunately, we can parallelize an entire stage of eviction by assigning one coprocessor to each of the $\sim 6D_sE_s$ entries involved. 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 first decide what will be evicted from all buckets, then update all involved buckets. As a consequence, eviction takes $O_C(B)$ 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 values $S \leftarrow \bigoplus_{m=1}^{q} v_m$ 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 1 (Oblivious Aggregation) *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*

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

It then outputs $z_m = x_m \oplus v_m$.

The untrusted server outputs $Z = \bigoplus_{m=1}^{q} z_m$.

§B contains proofs of correctness and security.

5.3 Dealing with Malicious Servers

Thus far, we have assumed an honest-but-curious server, but in the real world, if we distrust the server enough to use ORAM to hide our data access patterns, then it seems sensible to treat the server as fully malicious. Since the server acts as a relay between coprocessors, there are four broad classes of attack available to a malicious server: 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 [8]. *Data integrity* can largely be addressed by having the coprocessors MAC their outputs and verify the 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 (see §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; maintaining freshness in a distributed setting (§5.3.2) is even more challenging. 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 make sure that Z is always self-attesting. That is, it consists of a statement concatenated with a MAC of that statement's hash.

5.3.2 Ensuring Data Freshness

Attacks on freshness take two forms: **(1)** giving a coprocessor the same state with the same input, twice, and **(2)** giving a coprocessor the same state twice with different inputs. Equivalently, the server can give two coprocessors the same state and either same or different inputs. Both attacks can violate obliviousness.

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. We prevent attacks in the second class by uniquely numbering each request and associating each state with a single request; this ensures that given the same state with two different requests, a coprocessor will only act on the request associated with that state.

DETERMINISTIC COPROCESSORS. The server may give a coprocessor the same state with the same input, twice, hoping that the coprocessor will behave differently, and hence reveal information. To prevent this attack, we design all coprocessor operations to be stateless and deterministic; thus, making a coprocessor process the same input twice will result in the same output both times, so the server learns nothing.

In more detail, coprocessors keep no long-term state beyond their keys. Their operations are designed to be deterministic given the inputs and the scheme's parameters (e.g., N and B). When random choices are needed, we use pseudorandomly generated by applying a PRF to the inputs and the current request number. The only true randomness a coprocessor uses is for encrypting its output. Since our encryption scheme is IND-CPA secure, the server gains no information by making us encrypt the same content repeatedly with different randomness.

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

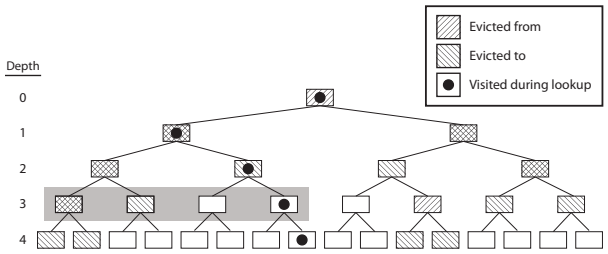

Figure 6: **Accessed Buckets.** *In each half of each level of the bucket tree, at most four buckets are accessed during a request: one evictor, two evictees that are children of evictors, and one that is accessed by the lookup for the specified address. Sometimes these overlap, particularly at the low 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.*

structure to the request that produced it. Thus, coprocessors will only act on data for request $r+1$ if the data 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, $O(BN\log N)$, and each request entails many modifications happening in parallel.

To cope with this scale, we use a collection of Merkle trees [19]. 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.

As an additional optimization, rather than validate the Merkle values for each entry for every operation a request entails, we validate the entry once when a request first arrives and generate a *freshness attestation* (a MAC) directly linking the entry to the current request number. After a request completes, we update the Merkle values for all of the entries affected by the request.

Merkle Tree Structure. 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 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; furthermore, as we will discuss in §5.5, it can quickly compute 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 $r+1$.

Creating Freshness Attestations. Before operating on an entry, a coprocessor must be sure it is fresh, i.e., that it corresponds to the state after the previous request. It can do so by getting an attested root hash of the corresponding Merkle tree, along with all the hashes along the Merkle tree path from the root hash to the entry's hash, and all the hashes of their siblings.

In practice, it is wasteful to do this separately for every entry, since there is a lot of overlap in the Merkle tree. Instead we assign one coprocessor to every Merkle root hash and have them generate attestations of needed Merkle-depth-1 hashes. Then, we assign one coprocessor to each Merkle-depth-1 hash for which we need attestations of their Merkle children. After $O(\log N)$ such steps, we have validated all entries we will need subsequently, either for lookup or eviction. Each coprocessor completes its work in $O_C(1)$ time, but unfortunately we cannot overlap this process for different stages since each is dependent on the last for determining the lookup path. Thus, the total time taken is $O_C(R\log N)$.

As a side effect of this process, we also generate attestations linking the previous request number to all siblings of all Merkle ancestors of nodes to be modified during this request. We use these below.

The number of coprocessors needed for the Merkle leaves is no more than that needed to process the entries, since a leaf represents an entry. The higher Merkle depths require far fewer coprocessors. In particular, at the Merkle depth at which Merkle hashes represent entire buckets, each Merkle tree contains at most four hashes that can change. This is because a Merkle tree represents buckets of only one depth and half, and at each bucket depth and half there is at most one evictor, two evictees, and one bucket used for lookup (Figure 6).

Handling a Request. When operating on an entry, a coprocessor expects a freshness attestation linking it to the current request. If the attestation validates, it performs the operation and outputs an updated attestation.

Since eviction and lookup can take place in parallel (see §5.2.6), we could potentially end up with two con-

flicting attestations for an entry touched by both operations (e.g., the nodes with both a dot and hatching in Figure 6). To avoid this, when an entry is touched by both operations, we make the coprocessor that performs the eviction operation on an entry authoritative, i.e., the one that outputs an updated freshness attestation that reflects both the eviction and the lookup. It is easy for the evicting coprocessor to do both, since an entry is only changed by a lookup if its address matches the address the user requested. Meanwhile, a coprocessor performing a lookup quickly determines, with two UHF evaluations, whether its block is subject to eviction. If so, it simply does not produce a new entry attestation.

Updating the Merkle Trees. After completing the processing of each request, the coprocessors must collectively compute new Merkle tree root values and attest to them being the result of that numbered request. This process is highly parallelizable. We can generate attestations for all Merkle tree leaves in parallel, then for all Merkle tree nodes at the next-higher depth in parallel, etc., until we have completed the roots.

A coprocessor assigned to generate a Merkle node attestation takes as input attestations for each of its Merkle children. For children that were changed during this request, that attestation binds to the current request number; for children that were unchanged, it binds to the previous request number. In this latter case, the coprocessor rapidly validates that this purportedly-unchanged child should not have been changed by the current request. If everything is valid, it computes the Merkle value for its assigned node and outputs an attestation linking it to the current request number.

To analyze the time to update the Merkle trees, we make the following observations. Each Merkle tree has Merkle depth $O(\log N)$. For any i, all Merkle nodes at Merkle depth i can be handled in parallel, even nodes from different Merkle trees and even different stages. And, finally, each coprocessor can handle a node in $O_C(1)$ time. Thus, the total time to update all the Merkle trees is $O_C(\log N)$ with $O(R\log^2 N)$ coprocessors. With $O(\log^2 N)$ coprocessors, it is $O_C(R\log N)$.

Summary. By doing all the work of validating hashes and updating the Merkle root in parallel, we make the total time to do so $O_C(R\log N)$ with $O(\log^2 N)$ coprocessors. This also allows coprocessors to read and update entries quickly, by verifying or generating only one MAC.

5.4 Fault Tolerance

We now consider mechanisms for making the service tolerate component failures. Since it is fairly well understood how to make untrusted components fault-tolerant through replication, we discuss only how to tolerate coprocessor failures.

As discussed in §5.3.2, our system is designed so all of the coprocessors, except the master coprocessor, keep no long-term state beyond N, B, UHFSeed, and keys. Thus, if a non-master fails, its functionality can be duplicated by having a new coprocessor install the ORAM code and join the ORAM service. Coprocessors do keep short-term state while they are in the process of performing a step of the algorithm, but if one fails during such a step, the server can perform it on another coprocessor.

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 coprocessor faults we want to tolerate. Each master instance independently keeps track, in its internal nonvolatile storage, of the last request number it has assigned. It will assign any higher number to any request, if asked by the server; it indicates this assignment by outputting an attestation linking the request's hash to the number. A request is considered definitively assigned a number when has been assigned the same number by a quorum of the instances, i.e., at least $f+1$ of them.

To prove that a request has been assigned the same number by a quorum of instances, a server needs not only the quorum of attestations but also an attestation to the list of instances in the master set. To get this, it supplies a list of master instance identifiers to the coprocessor that initially creates the ORAM service.

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.

Over the lifetime of the ORAM service, it is likely that $f+1$ coprocessors will fail. Therefore, it is useful to change, perhaps every one-hour period, the set of master instances to replace failed ones. In this way, we can cope with the loss of up to f coprocessors every period. We change the set of master instances via *delegation*.

To begin delegation, the server selects a new set of master instances. Then, it asks each of the existing masters to delegate its authority to the new set starting with the next request number r. A coprocessor performs this delegation if its highest number assigned so far is less than r. It delegates its authority by deleting its master instance and outputting an attestation of its delegation. This attestation 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. §C proves that delegation can

never cause two requests to be assigned the same request number.

5.5 Recommended Optimizations

We now describe some recommended optimizations.

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 `LookUpAndRemove` 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 `SatisfyUserObliviousRequest`. 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 `LookUpAndRemove`.

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. [26], 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 §D.

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 $\leq (1/2)\delta + (1/4)\delta + (1/8)\delta + \cdots = \delta$.

6 Implementation

To evaluate the performance of our parallel scheme, we implemented the base scheme described in Section 5.2 on Infineon SLE 88 cards. This includes parallel versions of all algorithms shown in Figures 18 and 19, as well as code to interact with remote users; it does not yet implement integrity or freshness checks. It is 1,927 lines of C, as measured by SLOCCount [33]. An additional 849 lines are used on the server side to communi-

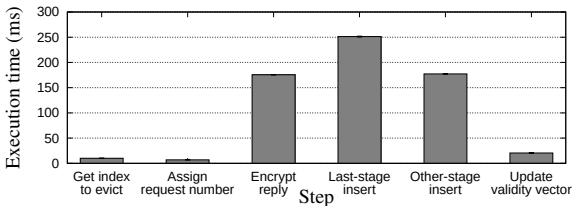

Figure 8: **Time to Perform Miscellaneous Algorithm Steps on the SLE 88.** *Times shown are for 1 KB block sizes. Integrity checks are not included. Error bars, generally too small to see, indicate 95% confidence intervals about the mean of 50 trials.*

cate with the cards. 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 function from UMAC [6] as the UHF.

To evaluate our system at scale, we built a simulator. It uses the measurements from our implementation to extrapolate the performance on thousands of machines, and also simulates server functionality. The simulator is 781 lines of Python, as measured by SLOCCount [33].

Our simulator can report the effect on latency of the optimizations that shorten the critical path and remove eviction from the critical path. However, it cannot determine the minimum number of coprocessors required for these two optimizations, because we have not yet built a scheduler that optimally assigns coprocessors to different types of tasks; this is future work.

7 Evaluation

In all experiments, unless stated otherwise, we use the following parameters: 2^{35} addresses; 10-KB records; 10,000 coprocessors; desired lifetime failure probability 2^{-80}; and ORAM lifetime 2^{50} requests. These are modeled after the map tiles workload.

7.1 Microbenchmarks

We evaluate how long different steps of our ORAM algorithm take on a real coprocessor, the Infineon SLE. We focus on the five 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: **First-stage lookup**, **Look up designator** (for intermediate stages), **Look up entry** (for the final stage), **Get evicted entry**, and **Post-eviction update**.

The left graph of Figure 7 shows the results of 50 trials for each step at each of four block sizes. Results for "look up entry" are not presented because they are close enough to "look up designator" that they would be hard to see. Also, the effect of integrity checks is not included, as our coprocessor implementation does not yet support them. We observe that, as expected, the dominant cost in all steps is proportional to record size; each of these steps takes ≈180–300 ms per KB.

The right graph of Figure 7 shows the predictions made by our simulator for these times. We see that they are in close agreement, and never off by more than 7%. Because our implementation does not yet support larger block sizes than 2 KB, or integrity checks, the rest of the evaluation uses our validated simulator to measure performance under various conditions.

For completeness, in Figure 8 we show the time taken by other algorithm steps implemented on the SLE 88.

7.2 Benefits of Parallelism

To evaluate parallelism, 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 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 9,000 sec to 4.7 sec, and for the map tiles workload, it reduces it from 140,000 sec to 31 sec. For the Flickr photos workload, it also reduces it substantially, from 14,000 hours to 2.9 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. With enough coprocessors, we could obtain better performance by splitting each image into smaller chunks and creating an ORAM over the first chunk from each image, an ORAM over the second chunk, etc., and look up all chunks in parallel.

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 can no longer have enough tasks to distribute among them. This occurs when we have about 28,000 coprocessors, enough to handle all $6\log^2(N)$ buckets involved in eviction in the final stage. Beyond this many coprocessors, we can use a few more to update all Merkle hashes at the lowest Merkle depth in all stages at once, but the benefits of this are slight since it does not involve movement of large data.

Figure 10 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 least until technology trends make HSMs with this bandwidth more affordable.

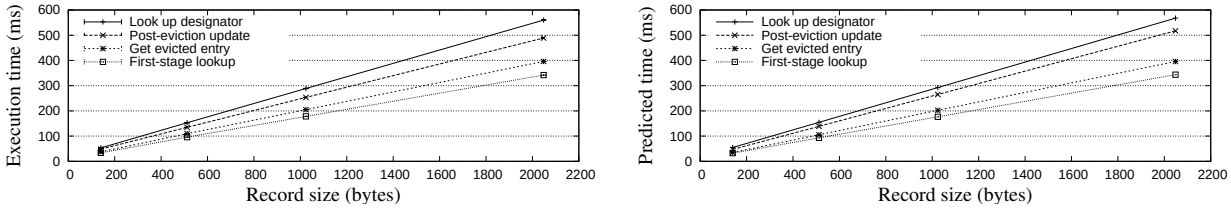

Figure 7: **SLE 88 Performance.** *The effect of block size on the time to perform the most performance-impacting operations in our ORAM algorithm. The left side shows actual measured performance, and the right side shows our simulator's predictions. "Look up entry" is not shown because it overlaps closely with "Look up designator". Integrity checks are not included. Error bars, generally too small to see, indicate 95% confidence intervals about the mean of 50 trials.*

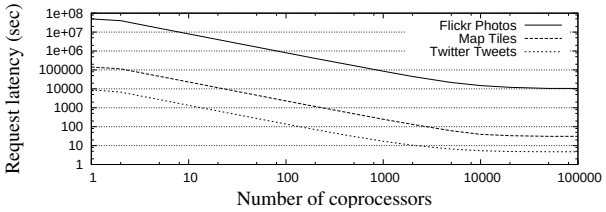

Figure 9: **Varying Number of Infineon SLE 88 Coprocessors.** *The effect of number of coprocessors on ORAM request latency for various workloads using simulated SLE 88 coprocessors. X-axis and Y-axis are log-scale.*

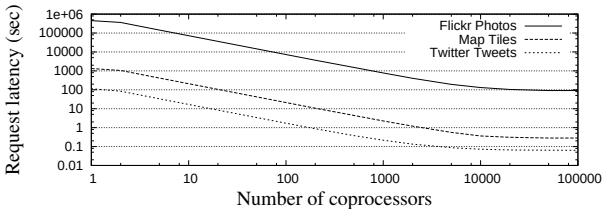

Figure 10: **Varying Number of IBM 4764 Coprocessors.** *The effect of number of coprocessors on ORAM request latency for various workloads using simulated IBM 4764 coprocessors. X-axis and Y-axis are log-scale.*

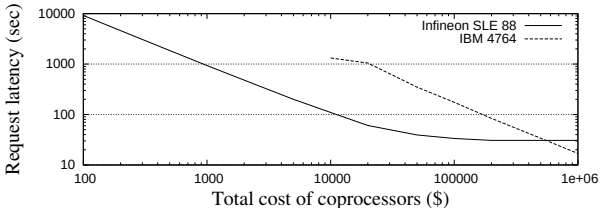

Figure 11: **Latency as a Function of Budget.** *As the amount of money available to purchase coprocessors increases, latency goes down. Beyond about $500,000, IBM 4764 coprocessors become more cost-effective than the inexpensive Infineon SLE 88 cards. X-axis and Y-axis are log-scale.*

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 11 shows the results. We see that for low budgets, the inexpensive Infineon SLE cards offer better performance per unit cost, since they are significantly cheaper. Only with more than about $500,000, when one can afford many more than 28,000 SLE cards and thus can get little benefit from buying more, do the IBM 4764 coprocessors become suitable for use in our parallel ORAM.

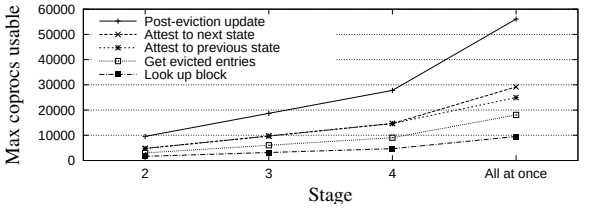

Figure 12: **Available Parallelism.** *How many coprocessors each step of the algorithm can make use of, for different ORAM stages. The stage "All at once" measures how many coprocessors a step can make use of when using an optimization that allows all stages to proceed in parallel (§5.5).*

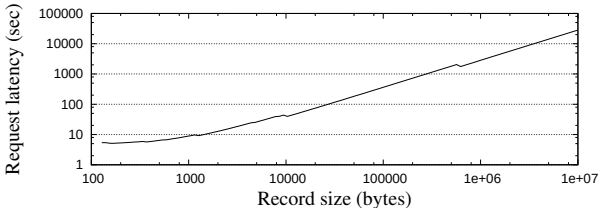

Figure 13: **Variable Record Size.** *Effect on parallel ORAM latency of varying record size. Simulation assumes 2^{35} records and 10,000 coprocessors. X-axis and Y-axis are log-scale.*

To understand which steps of the algorithm are most amenable to parallelism, Figure 12 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 post-eviction update, which updates all the entries of all evictor and evictee buckets, which is about $6\log^2(N)$ entries. In contrast, getting evicted entries only reads entries of evictor buckets, of which there are about $2\log^2(N)$. The step that can make the least use of them is lookup, which only updates all the entries of buckets along the designator path, which is about $\log^2(N)$ entries.

7.3 Effect of Record 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 records stored and the number of such records.

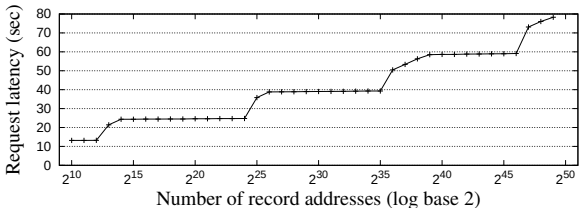

Figure 14: **Variable Record Count.** *Effect on parallel ORAM latency of varying number of addresses where records can be stored. Simulation assumes 10-KB records and 10,000 coprocessors. X-axis is log-scale and starts at* 2^{10} *records.*

Figure 13 shows the effect of varying record size on ORAM performance, with all other parameters set to defaults including coprocessor count 10,000. Overall, we see that below about 1 KB, the time taken is under 10 sec, and below about 16 KB, the time taken is under one minute. Thus, while ORAM is conceivably feasible for workloads like Twitter tweets, Facebook images, and map tiles, it is unreasonable for the Flickr workload's 5-MB record size.

Another thing to observe is that at low block sizes, below about 1 KB, the effect of constant overheads dominate, 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 2.8 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 14 shows the effect of varying number of addresses N on ORAM performance. We see that, unlike record size, record count has a less dramatic effect on request latency: in most cases, doubling the address count has no effect on performance. For some doublings, there is a large jump, due to the number of stages $R = O(\log_c(N))$ 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 record counts seen in our data sets, 2^{32} to 2^{36}, request latency is between 39 and 51 sec, but even data sets containing substantially more records than this would have comparable latency.

7.4 Cost of Defense Against Malice

Next, we evaluate the cost of defending against malicious servers in the Binary Tree algorithm. Recall that we must use variable and entry attestations to ensure data integrity, and ensure freshness by including request numbers. Because we cannot afford to re-attest the whole state on every request, we use Merkle trees as an optimization to enable freshness assurance. To evaluate the cost, we run experiments in which we turn off integrity and freshness checks.

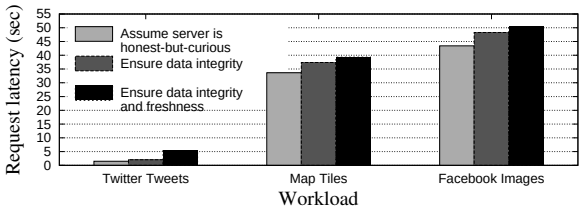

Figure 15: **Cost of Defense Against Malicious Servers.** *Increase in ORAM latency due to defenses against malice.*

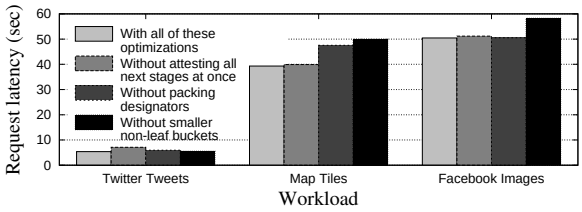

Figure 16: **Effect of Default Optimizations.** *Increase in ORAM latency when various optimizations we devised are removed. Simulation assumes 10,000 coprocessors.*

Figure 15 shows the results. We observe that 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 record size means that the cost of communication dominates, and thus integrity and freshness checks are significant. Time increases by 35% for integrity, due to the need to compute and transmit MACs; adding freshness increases it by a further 223%, primarily due to Merkle tree maintenance. For the other workloads, integrity checks increase latency by about 11% and freshness by an additional 5–6%. The more modest overhead is because the hashes involved are small compared to the 10-KB blocks that coprocessors must access.

7.5 Comparison to the Hierarchical Scheme

To emphasize the importance of amortized analysis, we simulate the traditional hierarchical algorithm [13, 23], which has $O(\log^3 N)$ amortized latency, but $\Omega(N)$ worst-case performance. Since the scheme is not parallelizable, for fairness we consider it to run on the fast coprocessor, the IBM 4764. With 2^{35} records 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 clearly unsuited to the data center.

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,

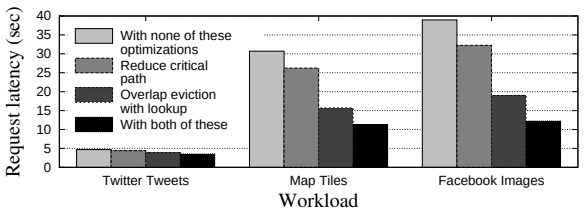

Figure 17: **Effect of Coprocessor-Intensive Optimizations.** *Decrease in ORAM latency when various additional optimizations we devised, which involve partially overlapping several stages, are added. Simulation assumes unlimited coprocessors.*

packing designators, and using smaller non-leaf buckets. Figure 16 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 24%. This is because this workload has a small record 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 17%. 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 21% and 13% 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 17. We see that the effectiveness of these optimizations is substantial for the workloads with large record sizes, and also significant for the small-record Twitter tweets. For Twitter tweets, reducing the critical path reduces latency by 6%; overlapping eviction with lookup decreases it by 15%; and doing both reduces it by 17%. For the others, reducing the critical path reduces latency by 15–17%; overlapping eviction with lookup decreases it by 49–51%; and doing both reduces it by 63–69%.

These large effects indicate it pays off substantially to use these optimizations when many coprocessors are available. Recall that Figure 12 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.

8 Conclusion and Open Questions

Using secure hardware to implement ORAM in the data center is one of the most practical approaches to private information retrieval. Nonetheless, existing ORAM schemes are not generally designed with data-center constraints in mind. We demonstrate that deploying ORAM in the data center creates new challenges and opportunities, including issues of scale, parallelism, throughput, maliciousness, fault tolerance, and worst-case performance. We show how to design an ORAM scheme to address all of these issues, and our evaluation suggests that these optimizations are needed to have any hope of scaling ORAM to data-center-scale workloads.

In keeping with the existing ORAM literature, we focus on improving request latency. However, data centers also care tremendously about throughput. While our scheme allows a few opportunities for improved throughput, e.g., via pipelined stage lookups or limited request batching, the existing ORAM paradigm seems inherently resistant to handling large numbers of requests simultaneously. Since replication is not feasible at this scale, new approaches or definitions are needed.

References

[1] 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)*, July 2007.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, Aug. 2010.

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

[4] 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.

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

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

[7] D. Boneh, D. Mazières, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Technical Report MIT-CSAIL-TR-2011-018, MIT, Mar. 2011.

[8] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Proc. of EuroCrypt*, 2001.

[9] 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.

[10] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

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

[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] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3), 1996.

[14] 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.

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

[16] A. Iliev and S. Smith. Protecting user privacy via trusted computing at the server. *IEEE Security and Privacy*, 3(2), 2005.

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

[18] 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.

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

[20] 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)*, Feb. 2011.

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

[22] 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.

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

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

[25] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proceedings of CRYPTO*, 2010.

[26] 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.

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

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

[29] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3), 2001.

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

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

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

[33] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: `http://www.dwheeler.com/essays/linux-kernel-cost.html`, Oct. 2004.

[34] P. Williams and R. Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[35] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *the ACM Conference on Computer & Communications Security*, 2008.

[36] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of ACM SIGCOMM*, Aug. 2011.

A Detailed Pseudocode

In Figure 18, we provide detailed pseudocode for the Binary Tree algorithm; the code makes use of the helper functions shown in Figure 19. To illustrate its use, 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.

Given a designator, we use `LookUpAndRemove` to look up the block. It looks through all of the entries in all of the buckets along the path from the root to the designated leaf. It also removes the block from the entry where it was found, so that a later access to the same address will not necessarily follow the same path. To remove the block without telling the server which entry is being invalidated, it updates *all* the entries it examines. This involves re-encrypting even the unchanged entries.

After looking up a block in stage s, we update it as necessary and insert it in the root bucket of stage s, this time with a new descriptor.

We must also run `EvictStage` on each stage, to evict one bucket from each side of each level of its tree. For each evictor bucket, we find a valid entry index, i.e., one corresponding to a non-dummy item, to evict. We then read the entry there to find what to evict. For each evictee bucket, we find the index of an invalid entry so we can store the entry its parent evicted; if the evictee is also an evictor, we use the same index for evicting and receiving. We then update all entries of all evictors and evictees, so the server cannot tell which few entries actually changed.

Additional Details. We denote by N_s the number of addresses served by the stage-s ORAM, so $N_R = N$. 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 $N_s = \lceil N_{s+1}/c \rceil$. $R = O(\log_c N)$, but is generally quite small because c is large; indeed, for the data sets from §4.1, $R \approx 4$.

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 [26]. Our analysis indicates we need $E_s = \log_2(MN_s(R-1)/\delta)$, and in §5.5 we show how non-leaf buckets can use even fewer. The root and depth-1 buckets need only one entry each since they are evicted on every operation.

```
GetNewDesignator (request # r, stage s):
  d̄ ← PRF(SymmetricKeȳ ‖ "NewDes" ‖ r ‖ s)  (mod 2^size)
  Return d̄

SetToDummyIfInvalid (designator d̄, request # r, stage s):
  d̄' ← PRF(SymmetricKeȳ ‖ "DumDes" ‖ r ‖ s)  (mod 2^size)
  If d̄ is invalid, set d̄ ← d̄'
  Return d      // at this point, it is no longer secret

FindValidIndex(stage s, bucket b):
  Read from storage the validity vector for stage s, bucket b
  Return index of first 1-bit in the vector, or 1 if none exist

FindInvalidIndex(stage s, bucket b):
  Read from storage the validity vector for stage s, bucket b
  Return index of first 0-bit in the vector

ObliviouslyReadEntry(stage s, bucket b, index ī):
  Result̄ ← ⟨0,0,0⟩
  For each index j from 1 to number of entries in b:
    Read entry Ē from storage for stage s, bucket b, index j
    If j = ī, then Result̄ ← Ē
  Return Result̄

ObliviouslyUpdateEntry(stage s, bucket b, index ī,
                       new entry Ē'):
  For each index j from 1 to number of entries in b:
    Read entry Ē from storage for stage s, bucket b, index j
    If j = ī, then Ē ← Ē'
    Write Ē to storage for stage s, bucket b, index j
```

Figure 19: **Pseudocode for helper functions.** *Overlined values must be hidden from the server.*

B Proof of Correctness and Security of Oblivious Aggregation

Correctness. Protocol 1 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 Protocol 1 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 $1 \le m \le q$, and another party (server) receives as output $S = \bigoplus_{m=1}^{q} v_m$. Then Protocol 1 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 $y_m \xleftarrow{R} \mathcal{V}$ 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: $v_m \xleftarrow{R} \mathcal{V}$.
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 $1 \le m \le q$.
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. ■

C 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 can-

not 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. ■

D 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 $E \geq \log_2[M(D-2)(R-1)/\delta]$. ■

```
SatisfyUserObliviousRequest (request # r, boolean ¯IsWrite, address ¯Address, block ¯BlockToWrite):
  For each stage s from 1 to R:
    If s = 1:
      ¯CurrentBlock ← the sole stage-1 block
    Else:
      EvictStage(stage s)
      ¯CurrentBlock ← LookUpAndRemove(address ¯Address, stage s, designator DesignatorToLookUp[s])
    If s = R:
      If ¯IsWrite, set ¯CurrentBlock ← ¯BlockToWrite
    Else:
      ¯CurrentDesignator[s+1] ← designator at offset ⌈¯Address/c^(R−s)⌉ mod c in ¯CurrentBlock
      DesignatorToLookUp[s+1] ← SetToDummyIfInvalid(designator ¯CurrentDesignator[s+1], request # r, stage s+1)
      Set designator at offset ⌈¯Address/c^(R−s)⌉ mod c in ¯CurrentBlock to GetNewDesignator(request # r, stage s+1)
    Write ⟨⌈¯Address/c^(R−s)⌉, ¯CurrentBlock, GetNewDesignator(request # r, stage s)⟩ as the new stage-s root bucket's entry
  Return ¯CurrentBlock

LookUpAndRemove(address ¯Address, stage s, designator d):
  ¯Result ← 0
  For each bucket b along the tree path indicated by d:
    For each index i from 1 to number of entries in b:
      Read entry ¯E from storage for ORAM stage s, bucket b, index i
      If ¯E's address field is ⌈¯Address/c^(R−s)⌉:
        ¯Result ← ¯E's contents field
        ¯E ← ⟨0,0,0⟩
      Write ¯E to storage for ORAM stage s, bucket b, index i
  Return ¯Result

EvictStage(stage s)
  Use a universal hash function to compute the set of buckets to evict BucketsToEvict
  For each bucket b in BucketsToEvict ∪ Children(BucketsToEvict):
    If b ∈ BucketsToEvict:
      ¯SelectedIndex[b] ← FindValidIndex(stage s, bucket b)
    Else:
      ¯SelectedIndex[b] ← FindInvalidIndex(stage s, bucket b)
  For each bucket b in BucketsToEvict:
    ¯EvictedEntry[b] ← ObliviouslyReadEntry(stage s, bucket b, index ¯SelectedIndex[b])
  For each bucket b in BucketsToEvict ∪ Children(BucketsToEvict):
    ¯ReplacementEntry ← 0
    If b ∈ Children(BucketsToEvict) and ¯EvictedEntry[Parent(b)]'s designator passes through b:
      ¯ReplacementEntry ← ¯EvictedEntry[Parent(b)]
    ObliviouslyUpdateEntry(stage s, bucket b, index ¯SelectedIndex[b], ¯ReplacementEntry)
```

Figure 18: Pseudocode for ORAM operations. R and c are public constants, as described in Figure 4. Overlined values must be hidden from the server. Some helper functions invoked here are described in Figure 19.