PrivateFS: A Parallel Oblivious File System

Peter Williams, Radu Sion, and Alin Tomescu
Network Security and Applied Cryptography Lab
Stony Brook University, Stony Brook, NY, USA
{petertw,sion,alin}@cs.stonybrook.edu

ABSTRACT

Privatefs is an oblivious file system that enables access to remote storage, while keeping both the file contents and client *access patterns* secret. Privatefs is based on a new *parallel* Oblivious RAM mechanism (PD-ORAM)—instead of waiting for the completion of all ongoing client-server transactions, client threads can now engage a server in parallel *without loss of privacy*.

This critical piece is missing from existing Oblivious RAMs (ORAM), which can not allow multiple clients threads to operate simultaneously without revealing intra- and inter-query correlations and thus incurring privacy leaks. And since ORAMs often require many communication rounds, this significantly and unnecessarily constrains throughput.

The mechanisms introduced here eliminate this constraint, allowing overall throughput to be bound by server bandwidth only, and thus to increase by an order of magnitude. Further, new de-amortization techniques bring the worst case query cost in line with the average cost. Both of these results are shown to be fundamental to any ORAM. Extensions providing fork consistency against an actively malicious adversary are then presented.

A high performance, fully functional PD-ORAM implementation was designed, built and analyzed. It performs multiple queries per second on a 1TB+ database across 50ms latency links, with unamortized, bound query latencies. Based on PD-ORAM, *privatefs* was built and deployed on Linux as a userspace file system.

Categories and Subject Descriptors

D.0 [**Software**]: General; E.3 [**Data Encryption**]

Keywords

Access Privacy, Cloud Computing, Oblivious RAM

1. INTRODUCTION

Access pattern privacy addresses a critical side channel leak present in many outsourced storage scenarios. Even on encrypted data, the sequence of *locations* read and written to storage reveals information about the user and the data. As a motivating example, consider a database management system running on top of an untrusted, encrypted file system. The file system, in satisfying requests for the transaction processor, learns semantic information about the transactions through the sequence of disk blocks accessed. If, for example, an alphabetical, encrypted keyword index is updated as an encrypted record is inserted, it can learn what keywords are present in the new record, *based only on the locations updated within the encrypted index*.

An oblivious file system enables a client to read and write without revealing the *access pattern* (which files are accessed, and any correlation between accesses). Without access pattern privacy, the act of accessing remote data leaks subtle information about the data itself, making it impossible to achieve full data confidentiality outside a narrow definition.

The oblivious file system presented here is built on the more general mechanism of Oblivious RAM (ORAM), which allows a client to read and write records into a database / memory hosted by an untrusted party, again hiding both the data and the *access pattern* from this untrusted host. Since the introduction of the first ORAM [2], approaches to increase query throughput have been sought and discovered. Nevertheless, practical constructions (and thus practical oblivious file systems) have remained elusive.

This paper introduces PD-ORAM ("Parallel De-amortized ORAM"), a collection of new techniques, applicable to a large class of existing ORAMs, to improve their performance and practical relevance by eliminating critical bottlenecks and drawbacks. PD-ORAM is then used as the block-level building block for *privatefs*, a parallel, oblivious file system.

First, the need for supporting parallel queries in ORAM is identified and satisfied. Existing ORAMs are characterized by a significant number of round-trips, typically $O(\log(n))$, per query (with some notable exceptions; see the related work discussion below). On wide-area networks, with mid to high latency, this imposes strict limits on the query response times as well as throughput. By supporting querying in parallel from multiple clients, PD-ORAM eliminates the effect of query response time on throughput, while presenting an opportunity for new multi-client scenarios.

A new de-amortization construction (converting an algorithm with an amortized bound into one with a worst-case bound) is then introduced to process queries simultaneously with database re-shuffling. Re-shuffling is an essential and extremely costly task which, in amortized solutions, completely blocks the server for extended time periods after a

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

certain number of queries. In PD-ORAM and other de-amortized solutions, instead of suspending queries to re-order data periodically, the server continuously re-shuffles the database in the background, in loose synchronization with querying, to guarantee minimal client latencies for both the average and worst case.

At an overview level, the parallelization technique consists of a round-trip-optimal and wait-optimal protocol to transform a single-client "period-based" ORAM into a multi-client parallel ORAM. A "period-based" ORAM is defined in Section 5.1 as an ORAM that operates on small batches of queries, with its data structure not sensitive to the order of reads or modifications within a particular batch.

A fully functional PD-ORAM implementation is developed and benchmarked at multiple queries per second on a terabyte database, the highest throughput to date on a medium-latency link. While the mechanisms described here can be directly applied to a large number of different ORAM techniques, PD-ORAM is based on the ORAM described in [19], but with de-amortized level construction, support for parallel queries, and a new, simpler, Bloom filter (BF) construction. PD-ORAM is then deployed and benchmarked in Linux to build *privatefs*, the first oblivious file system.

2. RELATED WORK

Oblivious RAM, introduced by Goldreich and Ostrovsky [2] is a primitive that provides access pattern privacy to *a single client* (or software process) accessing a remote database (or RAM). The construction provided by Goldreich and Ostrovsky (referred to as GO-ORAM) requires only logarithmic storage at the client; the amortized communication and computational complexities of this construction are $O(log^3 n)$. A discussion of GO-ORAM and recent extensions follows.

2.1 ORAM Overview

In an ORAM, the database is considered to be a set of n semantically-secure encrypted blocks (with an *ORAM key* held by the client) and supported operations are read(id), and write(id, $newvalue$). In GO-ORAM the data is organized into $\log_2(n)$ *levels*, as a pyramid. Level i consists of up to 2^i blocks; each block is assigned to one of the 2^i buckets (originally $\log_4(n)$ levels sized 4^i; levels sized 2^i are used here for simplicity) at this level as determined by a hash function. Due to hash collisions each bucket may contain from 0 to $k \ln n$ blocks.

ORAM Reads. To obtain the value of block id, a client must perform a read query in a manner that maintains two invariants: (i) it never reveals which level the desired block is at, and (ii) it never looks twice in the same spot for the same block. To maintain (i), the client always scans a single bucket in every level, starting at the top and working down. The hash function informs the client of the candidate bucket at each level, which the client then scans. *Once the client has found the desired block, the client still proceeds to each lower level, scanning random buckets instead of those indicated by their hash function.* For (ii), once all levels have been queried, the client re-encrypts the query result (so it looks different to the server) and places it in the *top* level. This ensures that when it repeats a search for this block, it will locate the block immediately (in a different location), and the rest of the search pattern is randomized. The top level quickly fills up; how to dump the top level into the one below is described later.

ORAM Writes. Writes are performed identically to reads in terms of the data traversal pattern, with the exception that the new value is inserted into the top level at the end.

Level Overflow. Once a level is full, it is emptied into the level below. This second level is then re-encrypted and re-ordered, according to a new hash function. Thus, accesses to this new *generation* of the second level will henceforth be completely independent of any previous accesses. Each level overflows once the level above it has been emptied twice. Any re-ordering must be performed obliviously: once complete, the adversary must be unable to make any correlation between the old block locations and the new locations. A sorting network is used to re-order the blocks.

To enforce invariant (i), note also that all buckets must contain the same number of blocks. For example, if the bucket scanned at a particular level has no blocks in it, then the adversary would be able to determine that the desired block was *not* at that level. Therefore, each re-order process fills all partially empty buckets to the top with *fake* blocks. Recall that since every block is encrypted with a semantically secure encryption function, the adversary cannot distinguish between fake and real blocks.

2.2 Recent Developments

Starting with [18] researchers have sought to improve the overhead from the polylogarithmic performance of the original ORAM. Williams et al. in [19] introduced a faster ORAM variant which also features correctness guarantees, with computational complexity costs and storage overheads of only $O(\log n \log\log n)$ (amortized per-query), under the assumption of $O(\sqrt{n})$ temporary client memory. In their work, the assumed client storage is used to speed up the reshuffle process by taking advantage of the predictable nature of a merge sort on uniform random data.

Recently, a new approach to speed up ORAM was revealed by Pinkas et al. [13], showing the applicability of the Cuckoo hash construction from [12]. Unfortunately, this was shown to leak access privacy information [3]. A similar, but secure, approach, allowing efficient item lookup while hiding success was then developed [3]. This approach has found continued utility in other solutions [4, 8].

Researchers have long recognized the utility of constructions with efficient worst cases; the first de-amortized construction followed shortly after the introduction of Oblivious RAM [11]. More recent solutions have also featured de-amortized constructions [1, 4, 8, 14]. These de-amortized solutions are mostly based on the same core idea as we describe here: constructing future levels in the background, while still querying copies of current levels. One exceptional ORAM [14] is naturally un-amortized (rather than de-amortized), performing a well-defined ORAM update on every query. These previous solutions do not apply directly to a Bloom filter-based ORAM; this de-amortization requires maintaining a delete log to delay level updates until after the corresponding shuffle.

A promising recursive construction technique is introduced in [16] under the assumption of $O(n \log n)$ reliable client storage. Using this storage, it promises to reduce the level construction cost while requiring only a constant number of online round trips. The clear drawback here is the assumed $O(n \log n)$ client storage—enough to keep track of the positions of all items, instead of querying recursively for them as in the previously described ORAMs. The authors

present the $O(n \log n)$ client storage assumption as not necessarily unreasonable, since a large block size means that the client only needs a fraction of the outsourced storage. Their alternative construction, requiring only $O(\sqrt{n})$ storage, recursively uses a $\log n$-round-trip ORAM to store this position map, but now incurs $\log n$ round trips per query.

Another notable alternative to the constructions of $\log(n)$ round trips is found in [17]; Ding et al. build an Oblivious RAM requiring only a constant number of online round trips. The main idea is to extend Goldreich's $\sqrt{n}$-solution [2] to store the recent query cache in client memory. The drawback is the significantly higher shuffle cost, since the entire database must be scanned once the cache is filled (e.g. after a period of $\sqrt{n}$ queries). Moreover, the amortized shuffle costs in Oblivious RAMs easily dominate the online costs both in theory and in practice [13]. Boneh et al. examine a similar construction [1], and formalize the security model.

Another constant-round-trip solution, based on the same core caching idea as [17] and [1], is introduced in [6], but the single-level format prevents de-amortization. The constant-round-trip claim depends on the assumption of $M = n^{1/u}$ client memory for a constant u. Nevertheless, important notions regarding optimal use of local memory in performing efficient oblivious external-memory sorts are introduced.

In addition to providing de-amortization for the construction in [19], PD-ORAM introduces a *general* approach to de-amortization, based on rearranging levels to safely allow querying during level re-construction. Different constraints of this base ORAM require new techniques, but the core idea remains similar to related de-amortization work. This de-amortization approach applies to any ORAM with a logarithmic number of levels. This includes a Bloom filter-based ORAM [19], the original Goldreich-Ostrovsky logarithmic ORAM [2], and recent cuckoo-hash based solutions [3], though we remind the reader that prior work [4, 11] has already de-amortized the latter two solutions.

A multi-client ORAM is introduced in [7]. Because the client state (aside from the secret key) is stored on the server, and scanned on every access, clients can take turns performing accesses. PD-ORAM takes this notion one step further: not only are the clients "stateless," but accesses are actually performed in parallel. That is, clients begin future accesses while other clients are still processing previous ones.

In this paper, we choose to instantiate our Bloom-filter based ORAM from [19]. Regarding the choice to use Bloom filter as an underlying ORAM, we are aware of two solutions with better than the Bloom filter ORAM's $O(\log^2 n)$ complexity: one provided by Stefanov et al [16] and one provided by Goodrich and Mitzenmacher [3]. Neither solution, however, can provide detection of misbehavior by an actively malicious adversary, which we deem to be an important property. Moreover, the techniques we introduce still apply to many existing ORAMs.

2.3 When Reality Hits

PD-ORAM is unique in providing a tangible de-amortized implementation. De-amortized constructions do not always lead readily to an actual implementation. Instructions such as "perform a chunk of work sized x", or "run the shuffle in the background while querying", are fine for establishing existence proofs, but can be devastatingly inappropriate in achieving an actual prototype due to large hidden constants. New techniques that address these hidden complications of de-amortization are presented in Section 5.3.

A randomized shell sorting network identified by Goodrich et al. [5] was subsequently employed to construct an Oblivious RAM [3, 13]. This sort procedure was used in the first design of PD-ORAM, but was found to result in too many disk seeks to make it usable on a even a medium-sized database. The randomized nature of the shell short guarantees that the order of item access appears non sequential and random, making efficient use of rotational hard disks difficult. The amortized cost of constructing the bottom level in a terabyte database is in the range of hundreds of disk seeks per query, already putting the implementation outside of the targeted performance goals.

To understand this in more detail, consider the construction of the largest level of a 1 TB database. This level contains at least 0.5 TB of data. For 10 KB blocks, this translates into 50 million blocks. The randomized shell sort makes $6k \log_2 n$ random passes across the database, incurring a total of $6kn \log_2 n$ seeks every n queries, where k influences the sort failure probability. For $n = 5 \times 10^7$, $\log_2 n = 23$, translating to $138 \times k$ disk seeks per query for the largest level alone.

For $k = 4$ as suggested in the original paper [5], this amortizes to 550+ disk seeks per query. Even for high-speed, low-latency disks with 6ms seek times, this becomes at least 3.3 seconds/query (in addition to any/all other significant network and CPU overheads)!

Multiple Disks. Of course this can be mitigated by using multiple disks at additional cost, in which case, the number of seeks for each disk is reduced (linearly in the added cost) as they may occur in parallel. However, increasing performance linearly in the added cost is not surprising nor desirable, and an efficient base-case construction should be found instead.

Unfortunately, ORAM imposes a unique sorting requirement that is difficult to satisfy using the randomized shell sort. This requirement derives from the fact that, to maintain privacy, the sort must succeed with overwhelming probability. Moreover, all sorts must be indistinguishable, eliminating the possibility of retrying in the case of failure. This is because observation of a sort failure translates into an advantage at distinguishing the permutation from random, which translates into a privacy leak.

While in other applications it may suffice to repeat the sort until it succeeds, when applied to ORAM, the sort parameter k must be chosen to guarantee success with overwhelming probability. In [5] the failure rate is determined experimentally; it is not obvious from the proofs what k is necessary to obtain a given acceptable error rate on database sized n. While the paper does prove that the probability of failure is negligible in k, which is sufficient for a construction existence proof, it is unclear what parameters can be chosen for a practical implementation.

It has been observed elsewhere that this sort is significantly less efficient than non-oblivious sorts. In particular, [13] opts to use a standard sort on the client for those levels that fit in memory. PD-ORAM uses the asymptotically equivalent merge sort, described in [20], that runs faster (requiring $\log_2 n$ sequential passes instead of $6k \log_2 n$ random passes), but requires $k\sqrt{n \ln n}$ blocks of local client memory.

Further, recently it was shown [13] that without a careful choice of Bloom filter parameters, the construction of [19] is

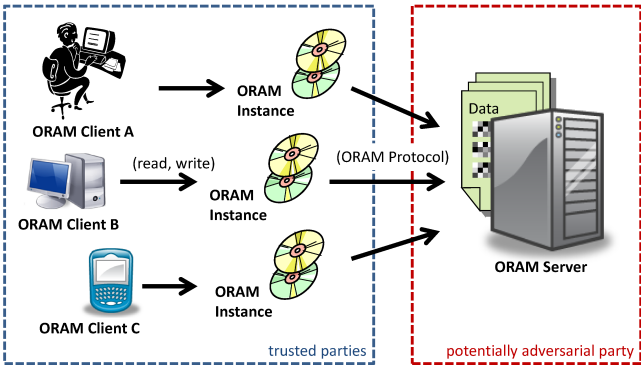

Figure 1: Overview: ORAM Clients access data obliviously through a simple ORAM Instance interface. The stateless ORAM Instances use an (untrusted) ORAM Server to store and retrieve the data obliviously.

susceptible to a leak via false positives in the Bloom filter construction. Avoiding this leak requires a number of hash functions proportional to the security parameter k. PD-ORAM uses this construction, choosing k to bound the false positive rate at 2^{-64} per lookup.

3. MODEL

An ORAM setup consists of three types of parties: ORAM Clients, who issue read and write queries, the limited-storage ORAM Instances, who satisfy these queries for their client while maintaining privacy, and the ORAM Server, who has plentiful storage and is willing to help the ORAM Instances, but is deemed untrustworthy (Figure 1).

The semantics used here correspond to existing ORAMs (read and write of fixed-size blocks is supported), with the addition of concurrency and multiple "Instances". Because of the introduction of concurrency, it is also *important to offer an atomic record-level test-and-set instruction* to ORAM clients. Analogous to the CPU primitive, it updates a record and returns its old value as a single atomic operation.

ORAM Client: a party who is authorized to issue reads and writes to the ORAM Interface. Data is accessed in "blocks", a term used to denote a fix sized record. "Block" is used instead of "word" to convey target applications broader than memory access (including file system and database outsourcing). Block IDs are arbitrary bit sequences. There are multiple clients. Each ORAM client has access to the following interface provided by a corresponding *ORAM Instance*: read(id): val; write(id, val); test-and-set(id, val): val.

Accesses are serialized in the order the ORAM server receives them, which guarantees each client sees a serialized view of its own requests. Among multiple clients, however, the ordering is not guaranteed to correspond with the time-ordering of incoming client requests.

ORAM Instance: a trusted party providing an ORAM interface to ORAM Clients. Instances are stateless but have access to the *ORAM key* (a secret shared among instances, enabling access to the ORAM) and address of the server.

The Instance-to-Server protocol details are implementation-specific (and typically optimized to the instance to minimize network traffic and the number of round trips).

ORAM Server: the untrusted party providing the storage backend, filling requests from the instance. The ORAM Instances communicate only with the server (not with each other). This corresponds to the idea that it is unreasonable to require clients to be aware and participate in every access of each other. Moreover, this reflects the notion that the adversary may have a complete view of the network.

The server is assumed throughout the bulk of this paper to be honest but curious; however, Section 5.5 details inexpensive adjustments that ensure "fork consistency" against even an actively malicious adversary. Defined in [9], fork consistency acknowledges that a malicious server can present different versions of the database to different clients, e.g., by not including updates from some of the other clients, but guarantees that each "fork" view is self-consistent. Moreover, clients will detect this behavior if they ever communicate with each other, or if the server ever attempts to rejoin the views. The server is allowed to know which client issues queries and when queries are issued. Implementations are assumed to be free of timing vulnerabilities.

Communication between the ORAM Client and ORAM Instance is assumed secured, e.g., with access controls on IPC if they are on the same machine, or with SSL otherwise. Communication between the ORAM Instance and Server is also secured, e.g., with SSL.

Notation. c = number of parallel clients, i = a level within a pyramid-based ORAM (for the smallest, "top" level, $i = 0$), k = security parameter, n = database size, in blocks, $\cong_c$ is equivalence modulo c.

4. PARALLEL QUERIES: A FIRST PASS

We now examine how to query existing ORAMs in parallel. The idea is to start with an ORAM on which it is safe to run non-repeating *unique* queries (targeting different records in the underlying database) simultaneously, and to build from this an ORAM which can also safely run *colliding* queries (targeting the same underlying data record).

Consider the ORAM in [2]. We can augment it to handle parallelism for sets of unique queries as follows.

- First, consider that as a client query searches across the database for a particular item, its accesses are by design indistinguishable from random.
- Second, by ORAM construction, different unique queries touch independent sets of database locations (because their coin-flips are independent).
- Now consider a set of multiple queries. Any of its reorderings results in accessing the same locations (provided each query gets the same coin flips), albeit in a different order.
- Thus, intuitively, it is safe for clients to submit unique queries simultaneously: the server sees an identical transcript independent of the queries.

The above privacy intuition only holds for non-repeating unique sets of queries, of course. Clients simultaneously querying the same item reveal this to the server, as their accessed locations are substantially similar. This raises the interesting question of how to guarantee query uniqueness over arbitrary incoming client query patterns, without revealing any inter-query collisions to the server.

Since the model requires the ORAM Instances to communicate only via the ORAM Server (capturing an adversary with a complete view of the network), one idea is to have Instances synchronize with each other via a server-hosted

data structure, in a way that prevents overlapping equivalent queries while still guaranteeing indistinguishability of all query patterns.

Strawman. The simplest approach to ensuring uniqueness is for each new client to examine ongoing queries. Finding an intersection, the client can then wait for ongoing queries to complete before trying again. This is fundamentally insecure, however, since the server learns when a query is being repeated by a later client, since the later client waits for the previous client. Further, correcting this by making all clients wait for all previous queries to complete before initiating their own query defeats the goal of parallelism.

Alphaman. To fix these issues, the server will help clients maintain an encrypted query log. Scanning this query log allows an Instance to identify simultaneously ongoing requests. In the case of overlap, a random unique query is executed instead. Once its query (real or random) is completed, the Instance reports its result back to the shared cache/result log. It then searches in the log for the result of the simultaneously ongoing overlapping query it had identified (if any). This guarantees that in either case the Instance gets to learn the result it seeks. We detail this below.

5. ABSTRACTIONS AND SOLUTIONS

We now outline the properties required of the underlying ORAMs to allow parallelism and de-amortization, and then provide constructions.

5.1 Parallelism Abstraction: "Period-based"

The main idea is to run queries simultaneously *between* reshuffles. We are limited by the size of the top level: this is the number of appends that can be performed before a shuffle is required. Thus, we designate the maximum parallelism *and* the size of the top level as c. Subsequent levels will be sized $c2^i$ for this reason.

DEFINITION 1. *A* **period-based stateless ORAM Instance** *is an ORAM that performs a series of c queries between each shuffle. The transcripts of unique queries within a period are independent of their order. Previously executed queries are scanned from a server-stored cache sized c, triggering a fake lookup instead.*

Insight. The goal of this abstraction is to capture the fact that the underlying ORAM already supports up to c simultaneous *unique* queries. Specifically, the ORAM runs in periods of queries over which the transcripts of unique queries are independent (e.g., the original ORAM [2] satisfies this). For a given period between reshuffling (which lasts several queries, until the top level overflows), and choosing the random number generator coin flips for two unique queries ahead of time, the transcript for each query is the same regardless of the order they are run. The transcripts of two identical queries, however, are inter-dependent, since the first-to-execute query necessarily searches farther down in the database than the second query, as the item is moved up to the top level once the first query completes. The second-to-execute query finds the item immediately at the top, and thus the remainder of the search is random.

5.2 Parallelism Construction

The motivation behind this construction is to minimize waiting, while guaranteeing serializability. The basic assumption about the underlying ORAM is that its data structure supports simultaneous querying for unique blocks. Then, the *ultimate purpose of this protocol is to make sure all the simultaneous queries request different items.* The challenge is to juggle this uniqueness requirement (in the presence of colliding queries from different clients) with the requirement that the server not learn any inter-query correlation.

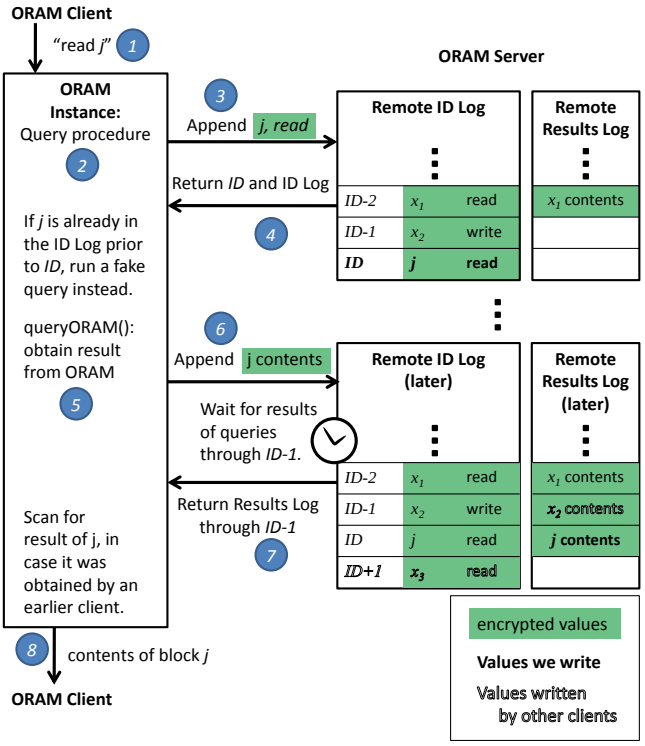

Figure 2: Parallel query protocol

One solution is to obliviously guarantee uniqueness of queries using an append log in combination with a results log. Clients will still need to wait for previous queries to complete before outputting a result, but now there is no requirement of blocking on other clients during query execution. This increases throughput significantly (Figure 2):

1. The Client issues a query to its ORAM Instance.
2. The ORAM Instance generates the request, consisting of the block ID, and a bit indicating whether this is a read or a write/insert. Test-and-set operations are identified as writes.
3. The request is encrypted and sent to the server.
4. The server appends this to the encrypted query log (analogous to the top level), and returns a sequential query ID, together with the query log, containing all queries since the top level was last emptied.
5. The ORAM Instance interactively queries the underlying ORAM. If the query was already in the query log (i.e., from another running client), the instance runs a dummy query instead.
6. The ORAM Instance sends its result, encrypted, back to the server, which appends it to the query result log.
7. The ORAM Instance reads the results log up to its own entry (only interested in previous clients' results), in case the current query is accessing a block that was previously read or written. This is the only step that requires waiting for the earlier queries to complete.

8. The ORAM Instance returns to the ORAM Client the result (obtained from the database, or the result log).

ORAM Instances wait for the entire result log prior to that query's registered location to complete before returning to the Client. However, in many cases the result will already be obtained. If it is assumed that the adversary cannot observe at what point the ORAM Instance returns to the client, then it is safe for the ORAM Instance to return its value sooner.

Server network traffic over a period of c parallel client queries is quadratic in c, since each Instance needs to be aware of what each simultaneous Client is doing[1]. The number of parallel clients that optimizes total throughput is thus a function of network bandwidth, latency, and database size.

On the one hand, for c clients, and a database of n blocks, the sequence of $\log_2(n)+3$ round trips per query imposes a network-latency based maximum query throughput of

$$\frac{c}{((\log_2 n)+3)\times \text{latency}}$$

On the other hand, the cost of supporting multiple clients (quadratic in c over c queries; linear in c per query), and the online data transfer cost of $\log_2 n$ blocks, impose a server bandwidth based maximum throughput:

$$\frac{\text{bandwidth}}{((c-1)/2+\log_2 n)\times \text{blocksize}}$$

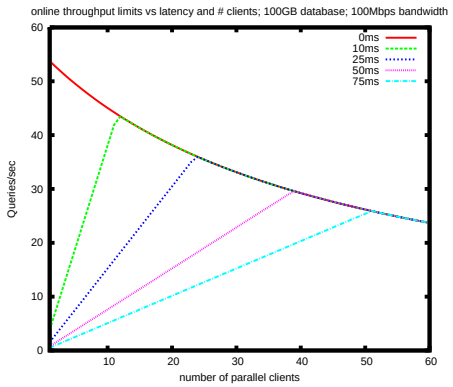

Figure 3: Upper bound on online query rate of a 100GB database (blocksize $= 10^4$ and $n = 10^7$) and assuming a 100Mbit (12.5 MB/sec) network link. The plot is shown for various network round trip times, from 0 ms through 75 ms. Single clients, incurring $\log_2 n$ round trips per query, are tightly bound by the round trip time. Adding more parallel clients increases this throughput linearly, up to the point where bandwidth limits from the query log traffic take over.

To find the optimal number of clients c for a given configuration the lower of these upper bounds needs to be maximized. This relationship is plotted in Figure 3 for a representative setting.

[1]To disseminate this information in a more network-efficient manner, e.g., by only requesting the log entry the client is interested in (instead of the entire log) a PIR protocol may be deployed. This results in a tradeoff between server bandwidth and server CPU time. This tradeoff is mostly unfavorable [15], yet recent results [10] indicate that it may become feasible in the near future.

After c queries execute, the query log is converted into the ORAM top level. Queries are thus applied to the database, in the ORAM-sense (Figure 4). This is where the requirement of a "period-based" Instance is necessary.

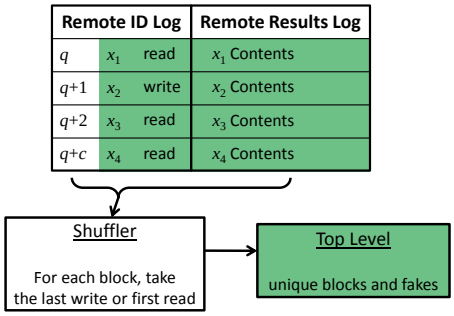

Figure 4: Reconciliation of the query log into the new top level. After a period of c queries, the query log is shuffled and becomes the top level. Regarding the query log hosted set of ids, consider that a block can only be read once from the ORAM during this period. Thus, the correct value for a given block is either the last write of this period, if there is one, or otherwise the first read.

In the process, the shuffler consolidates the query log. A single block may be accessed multiple times by different queries, but is placed only once back into the database. The value chosen for a given block is the last write, if there is one, or otherwise the first read. Recall that subsequent reads are associated with fake results.

Properties. It is time to informally define two properties: *optimality* and *query privacy*.

THEOREM 1. *In any model in which the server can associate all visible read/write data accesses corresponding to a given query, hiding access patterns in a "non-simultaneous ORAM" requires waiting for the results of all previous queries ("wait-optimality").*

PROOF. Take any "non-simultaneous" ORAM that requires serial execution of queries for the same underlying block. In such an ORAM, repeating the query before the previous query has completed would result in overlap between visible data accesses, and would allow an adversary to correlate these queries.

Thus, instead of attempting to repeat a previous query, clients must obtain the result from it once that query has finished executing.

To make all query sequences indistinguishable to the server, even in a parallel ORAM, clients still need to wait for the prior queries, as if their query depended on each of the previous queries (lest they reveal which query they are waiting for, if any). □

Theorem 2 shows that running parallel queries in this model is safe. The intuition is as follows. In the parallel case, client behavior from the perspective of the server is identical for each new query instance regardless of whether and how often the same equivalent query appears earlier in the log. For every query, the server only sees a semantically secure encrypted *append* operation to the query log, a *query* to the underlying ORAM, and a *scan* of the results log up to that point.

The only difference now is that transcripts of the different queries are interleaved, but otherwise contain the same accesses as when executed in a traditional ORAM. This is so because in the parallel case, if a client queries for a block that is already queried for by a simultaneously ongoing query, the client's ORAM Instance will instead issue a fake query—which is what it would have done anyway in the traditional ORAM had it found the query result at the top level. Thus, from the server's point of view, the transcripts contain the same (random looking) accesses.

Further, query transcripts are independent of their query and, without knowledge of the secret ORAM key, indistinguishable from random, as required by Definition 18 of traditional ORAM [2]. Then, an advantage at distinguishing the new transcripts translates into an equivalent advantage at distinguishing the underlying ORAMs if parallelism were not enabled.

THEOREM 2. *Existence of an adversary with non-negligible advantage at violating query privacy (as in Definition 18 [2]) in a parallel ORAM implies existence of an adversary with non-negligible advantage at violating the privacy of the underlying single-client ORAM ("query privacy inheritance").*

PROOF. Take an adversary A with advantage ϵ at correlating queries in a parallel ORAM based on an underlying ORAM O. We now construct an adversary B with equivalent advantage at correlating queries in O.

B simulates adversary A on every query. Since the interleaving information is publicly known, it includes this information in the transcripts it uses to simulate A. It appends random information to the query log contents for A. These query log contents give A no additional non-negligible advantage; otherwise a distinguisher could be built distinguishing the semantically secure encryption function output from random.

B then outputs the guesses and requests provided by A, and obtains the same advantage as A. □

5.3 De-amortization abstraction / construction

DEFINITION 2. *A* **level-based amortized ORAM Instance** *is an ORAM that searches levels recursively, appending the result back to the first level. Privacy results from the property than an item is sought at a particular level no more than once between two consecutive shuffles of that level.*

De-amortization techniques need to deal efficiently with the level constructions resulting from overflow of the top levels. Their goal is to arrange the levels such that they can be queried while the items are simultaneously being inserted and re-shuffled into new levels. That is, instead of suspending querying to wait for shuffling to proceed, a new level must be available as soon as it is needed.

The main idea is to provide pre-emptive shuffling. Rather than waiting for querying to complete before shuffling a level, its transformation into a new level begins as soon as a level is constructed, and right as its querying begins.

To allow this, we duplicate a level into two copies: a read-only variant that is used in the querying process, and a writable variant which is dynamically updated into the new generation of this level. The read-only copy is discarded at the end of the period.

Level De-amortization. Consider the de-amortization of a single level. In a traditional ORAM, a level is reconstructed by combining into it the above level that has filled up and now overflows (Section 2.1). This necessarily stops the query process for its duration.

To de-amortize this, when beginning its construction, instead of pausing queries and waiting for the construction to finish, querying can continue via the read-only level copy while a new generation is produced into the writable variant. Critically, during this process, existing levels can overflow into a fresh, empty, replacement for this level. (Figure 5).

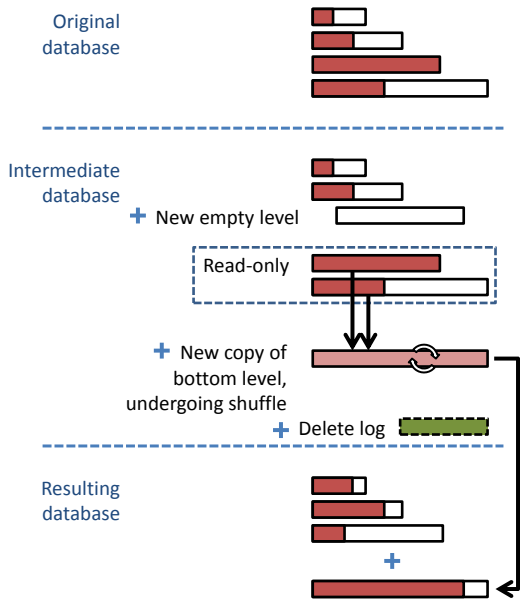

Figure 5: Background construction of a single level. The top section represents the initial database state; the middle section shows the database state during the process of constructing the new bottom level; the bottom section shows the resulting database state. In this scenario, the third level is full, so it needs to be combined with the fourth level. Read-only copies of those two levels are made, and can be queried while the combination is occurring. Simultaneously, overflows from the top level are placed into the a replacement third level. All five levels are accessed during queries at this time. Once the construction of the bottom level is complete, querying resumes with this new bottom level replacing the two read-only levels (which contain the same items).

Delete Log. The second change is to delay level updates until the end of the shuffle. Some ORAMs avoid the complexity of reconciling multiple versions of an item (and in the process, reduce storage overheads) by deleting blocks from the levels where they are found. In the de-amortized construction, this is not possible, since the queried level copies are now read-only. Instead, these changes are appended to an update log. Items marked for deletion in this log are now deleted by the server before the *next* shuffle of the level. Not all ORAMs modify the pyramid structure during queries; some [2] do not need such modification. This de-amortization protocol requires that those changes that are made can be applied *after* the level has been reconstructed.

Details. A level at height i, containing $m = c2^i$ items, is queried m times. At the end of the mth query, the shuffle

will have completed, and the remaining items from this level are now in a new level.

Each level is a data set, completely specified by its height i, the sequentially increasing "generation" j at that height, and a one-bit marker for the odd generations. The two levels at an even generation j, denoted by $i.j$ and $i.j.*$, are combined to produce a level at the next height $i+1$, generation $j/2$. Those levels at a height i with an odd generation j are reshuffled to produce level $i.j+1.*$.

The levels currently reshuffled are the ones queried – at any one time there are either one or two active levels (queried and being shuffled) at each given height, for $i \leq \log_2 n$.

The above construction allows de-amortization of the construction of all the levels except the top level. De-amortization of the top level appears to be possible using a rotating query log, but is left as future work (since its impact is minimal as the top level is very small).

If based on an underlying stateless period-based amortized ORAM, the result is a stateless period-based de-amortized ORAM, suitable for parallelization.

THEOREM 3. *A polynomially bounded adversary has no non-negligible advantage at guessing the access pattern based on observing the transcripts of a de-amortized ORAM.*

PROOF. For simplicity, the proof is given for BF ORAMs (Section 5.4) but property should hold for any secure underlying level-based ORAM. The output of the underlying BF ORAM level construction process is shown in previous work to be a randomly ordered set of stores (ID,value pairs). Both the ID and value are opaque; that is, the server cannot distinguish them from random. The only other time the server sees this opaque ID is when the ID is retrieved later on. The same opaque ID is never retrieved twice. BF locations are also accessed in a manner uncorrelable to the access pattern, as the locations corresponding to any ID are chosen with a pseudo-random number generator. □

THEOREM 4. *If the underlying level-based ORAM destroys inter-item correlations on shuffle, then existence of an adversary with non-negligible advantage at violating query privacy in a de-amortized ORAM implies existence of an adversary with non-negligible advantage at violating the privacy of this underlying level-based ORAM ("privacy inheritance").*

PROOF. A de-amortized ORAM operates equivalently to the underlying ORAM, with two differences. First, the level access structure is different—there are up to twice as many levels—but the privacy-preserving property that no item is requested twice under the same (opaque) identifier is the same. Second, items are not deleted until after the shuffle (instead of before), but this provides no additional information as long as the level construction process is also opaque.

This is the only new information given to the adversary. Privacy is shown by reducing to a property of ideal permutations: uncovering a portion of a permutation does not reveal anything about the remaining portion. Since the underlying level-based ORAM selects the level permutation according to an ideal secret permutation, this follows trivially: all arrangements of the elements in the "still secret" portion are equally likely. It constitutes a secret permutation in itself.

This results in an ordering and labeling indistinguishable from random. □

Storage overhead. Since the client storage required to shuffle level i is less than or equal to the space required to shuffle the next larger level, $i+1$, de-amortization requires an extra factor of client storage upper-bound by $\log_2 n$ (since all $\log_2 n$ levels are being shuffled at once).

Communication overhead. This construction requires querying up to twice as many levels simultaneously. However, since it is known in advance that the item won't show up in both levels, this querying can be done in the same number of round trips.

Shuffling overhead. The amount of work performed per query is roughly similar. Additional overhead may stem from shuffling being done *before* level items are removed. But since only half of the items would be removed anyway, for ORAMs where the level i construction cost is $sm\log_2 m$ for constant s and $m = 2^i$, this keeps the new overhead for shuffling within a factor of $\frac{s(2m\log_2 2m)}{s(m\log_2 m)} = 2 + \frac{1}{\log_2 m} < 3$.

5.4 BF ORAM with logarithmic memory

We now detail the final piece of the construction: the underlying base ORAM mechanism. For illustration we take our ORAM [19] that separated each ORAM pyramid level into two data structures: a hash table of items, indexed by unique random IDs, and an encrypted BF to check whether an item is stored at a given level. Querying this ORAM proceeds analogously to the standard pyramidal ORAM model—starting at the top, searching downward until the item is found, querying randomly from then forward. However, instead of scanning a bucket sized $O(\log n)$ at each level, the encrypted BF is checked, requiring only a single item retrieval from each level.

At reshuffle time, in [19], BFs for each level were built securely by the client using a somewhat complex procedure based on an oblivious scramble of a list representation of positions to set, followed by bucket sort. This allowed costs under $O(\log^2 n)$, requiring $k\sqrt{n \ln n}$ client storage.

We now introduce a technique resulting in an $O(\log^2 n)$ BF-based ORAM and requiring only *logarithmic* client storage. Instead of using the "bucket sort" method that builds $\sqrt{n}$-sized chunks of the encrypted BF, we construct, then sort, a server-stored list of the BF segments and indexes:

First, in a single pass over the encrypted items of the level undergoing construction:

- build an encrypted, server-stored, list of the positions that need to be set in the encrypted BF
- divide the BF into segments of size $O(\log n)$
- append one encrypted segment identifier for each possible segment to this list, padded so the server cannot distinguish segment identifiers from BF positions.

Second, obliviously sort this entire list [5, 19], with segments distributed among the positions, directly following the positions they belong in.

Third, in a single pass over this sorted list:

- output one BF segment for every list element: for each segment identifier, output an encrypted segment, with the appropriate positions set. Recall that these positions immediately preceded this segment identifier in the sorted list.
- for each position encountered in this list, output a dummy empty segment.

Fourth, obliviously sort the resulting list by segment identifier, with the dummy segments at the end.

Fifth, truncate off the dummy segments from this list. The result is the encrypted BF.

The advantage of this approach over [19] is that using a logarithmic-space $O(m \log m)$ oblivious sort [5] provides a logarithmic-space construction still running in $O(\log^2 n)$ time. As discussed, employing this as a de-amortized ORAM brings the client storage requirement to $O(\log^2 n)$.

This construction process is secure, provided (a) the oblivious sort is private, (b) the server cannot distinguish an encrypted dummy segment from an encrypted BF segment, and (c) the server cannot distinguish an encrypted segment identifier from an encrypted BF position. If these three conditions are satisfied, then every run of this process over a set of m items appears identical to the server, regardless of the positions in the BF being set.

The PD-ORAM implementation assumes $k\sqrt{n \ln n}$ client storage to employ the Oblivious Merge Sort instead of the disk-seek intensive less efficient randomized shell sort. Nevertheless, this simpler BF construction process is employed instead of the $k\sqrt{n \ln n}$ variant.

5.5 Security against malicious adversaries

Ensuring security against a malicious adversary first requires an underlying ORAM providing these guarantees. Fortunately, the BF-based ORAM of [19] provides such a mechanism. Second, maintaining security in a parallel setting requires clients to test that they all see a consistent view. This is achieved using a hash tree over the set of all previous queries. Whenever a client performs the top level shuffle, it is responsible for updating this hash tree and attaching the root value with a MAC to the new query log. This entails hashing the current query log, appending it as a new leaf node to the hash tree, and recomputing hash values along the path from this new node to the top.

Second, whenever a client performs a query, it performs one additional operation: it verifies that the last query it performed is included in the hash tree, whose root corresponds to the value attached to this query log. This is done by verifying all nodes in the hash tree adjacent to the path from the root to its last request.

These operations will not detect a forking / split universe attack, unless out-of-band communication is available, or clients perform periodic accesses to ensure they are still operating in the same view as other clients. However, this solution has the property that once the server forks the universe into multiple views, it cannot rejoin the views without being detected by the clients (by hash value disagreement).

The PD-ORAM implementation analyzed in the following sections, however, assumes an honest but curious adversary.

6. EXPERIMENTS AND ANALYSIS

Amortized Measurements. A significant challenge in measuring the performance of any amortized system is ensuring the trial captures the average performance, not just peak performance. This is complicated by the requirement of running trials for periods that are too short to encompass the full period over which the amortization is performed. For example, at even several queries per second, the reconstruction of the lowest level of a terabyte database is amortized over a period on the order of weeks or longer.

De-amortizing the level construction provides the opportunity of improving measurement accuracy. The challenge remains, however, of ensuring the de-amortized background shuffle proceeds proportionally to the query rate: the de-amortization is perfect when the new level construction is completed at the instant it is needed by a query. Inaccuracies in this rate synchronization will affect the measured results, since measured query throughput of a short period might be higher or lower than the sustainable rate.

To avoid this effect, PD-ORAM maintains *progress meters* for level construction, allowing queries to proceed when every level is proportionally constructed. The level constructions processes are also suspended when a level gets too far ahead of the current query. This keeps querying and level construction smooth, minimizing worst case latency.

Proper de-amortization: Theory vs. Reality. Performing proper de-amortization proved a non-trivial systems challenge. Research solutions, such as [4], and PD-ORAM (Section 5.3) express de-amortization in terms such as "perform the proportional amount of work required", or "perform the next $O(f(x))$ accesses." While these terms suffice for proving existence of a de-amortized construction, programming models do not typically provide this type of abstract control. PD-ORAM achieves this control by metering progress over the construction of individual levels. Since the level construction involves different types of computation across the client and server, accurate progress metering required splitting level construction into tasks whose progress can be reported over time. Moreover, this metering uses experimentally determined values to identify what portion of the level construction corresponds to which subtasks.

As an aside, suspending the level construction when it outpaces the queries proved critical on the larger database sizes. The sheer number of requests being sent from the ORAM Instance to the ORAM Server for construction tended to starve the requests of actual queries (much fewer in number), causing the query rate to drop quickly as more levels were introduced. This behavior was corrected by forcing level construction to remain proportional to query progress: this keeps the individual query rates much closer to average.

Since it is impractical to repeatedly running trials over the entire (up to 1TB) measured epoch, the database for these trials is first constructed non-obliviously on the server via a specially designed module. The items are inserted randomly so that the final result mirrors an oblivious construction (as would occur from a sequence of write queries).

6.1 Setup

PD-ORAM is written in Java. Clients run on quad-core 3.16GHz Xeon X5460 machines. The server runs on a single Quad-Core Intel i7-2600K Sandy Bridge 3.4GHz CPU, with 16GB DDR3 1600 SD-RAM and 7x2TB HITACHI Deskstar 7200RPM SATA 3.0Gb/s disks (RAID0 / LVM).

All the machines share a gigabit switch. Network latency is shaped by forcing server threads to sleep for the desired round trip duration upon receiving a request. This allows simulation of link latency without capping link bandwidth.

The implementation uses a BF with 8 hash functions, and 2400 bits of space per item which allows an efficient construction within the false positive rate of 2^{-64} per lookup. The resulting BF constitutes roughly 25% of the total size of the database records.

Optimization. Rather than optimizing the BF size required to obtain this error rate by using a larger number of hashes, as suggested in [13], PD-ORAM uses larger BFs

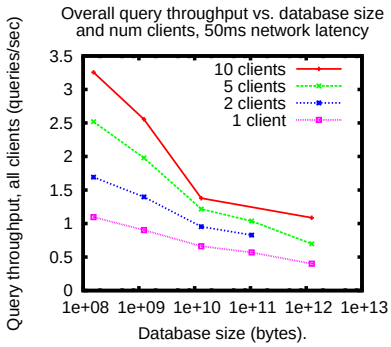

Figure 6: Query throughput vs. data size for varying number of clients x-axis is log scale. Each point is sampled over 3000 queries.

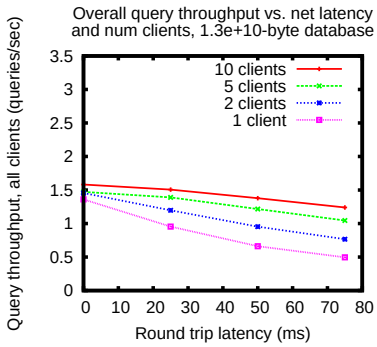

Figure 7: Query throughput vs. network latency for varying number of clients. Each point is sampled over approx. 3000 queries.

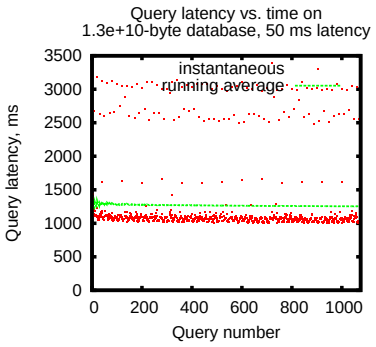

Figure 8: Individual query latencies + running averages of a single client. With perfect de-amortization, all queries would require the same amount of time.

with fewer hashes, to minimize item lookup disk seeks while obtaining the same error rate.

6.2 Experiments

One main goal of the experiments is to understand the interaction between network performance parameters and the parallel nature of PD-ORAM.

Size + clients vs. query throughput. Figure 6 plots the effect of database size and client parallelization on overall query throughput. Fresh databases were used for all trials to prevent dependency of the measurements on the order of the trials, except for the 1TB trials, where this proved impractical. Even though individual query latency increases with higher resource contention, the benefit of parallelization are obvious: significantly higher overall throughputs.

Clients + network latency vs. performance. Figure 7 plots the effect of parallel clients and network latency on overall query throughput for a fixed database size. The premise of this measurement is that parallelization becomes more important as network latency increases.

De-amortization optimality. Figure 8 plots the observed latency of individual queries vs. time on a growing database. With perfect de-amortization, all queries would require the same amount of time. Most queries take around 1200ms; a fixed lower limit is imposed by the network latency. The bands at 2600ms and 3100ms reflect the construction of the top level, which is not de-amortized.

Progress metering. To validate the accuracy of the progress metering, Figure 9 shows the reported construction progress of a single level as sampled every 5 seconds. Strict de-amortization and querying is disabled to avoid cool-down periods when construction has progressed farther than is needed, and to ensure measurement of its progress only.

6.3 Impact of disk latency

The experiments were repeated (for database sizes up to 300GB) in a different setup, in which the server was run on dual 3.16Ghz Xeon X5460 quad-core CPUs and *six 0.4 TB 15K RPM SCSI (hardware RAID0)* disks.

This configuration surprisingly outperformed the setup above by a factor of 2x in most trials. The primary advantage is the superior seek time on the server disks, so the markedly different results suggest that server disk seek costs play an important role in overall performance. This was somewhat surprising, since the level construction mechanisms were designed specifically to minimize disk seeks (with the hash table insertion being the only random-access operation during level construction, requiring an average of 2 random writes per insert). The rest of the level construction simply requires reading from one or two sequential buffers, and writing out sequentially to one or two.

The culprit is most likely the de-amortization process, which constructs different levels in parallel, and in effect randomizes disk access patterns. While individual level construction is mostly limited by sequential disk throughput, running many of these processes in parallel across the same file system results in disk seeks even in the sequential access regions, resulting in a much lower overall disk throughput.

SSDs. Several software and hardware solutions present themselves. Better data placement would split data in a more efficient manner across the available disks (instead of using a RAID configuration), to allow the sequential nature of each level construction process to transfer to sequential disk access. Obtaining optimal throughput in this manner would require a relatively large number of disks. Further, the use of more expensive (but quickly dropping in cost) low-latency solid state disks (SSDs) would be a simple hardware solution to eliminate this performance bottleneck. It remains to be seen however, whether the sustained random write performance degradation plaguing current SSDs does not constitute a bigger bottleneck in itself, as preliminary throughput experiments on several recent 128GB Samsung SSDs with 2011 firmware updates seem to suggest.

7. AN OBLIVIOUS FILE SYSTEM

ORAM lends itself naturally to the creation of a block device. Due to existing results' impractical performance overhead this has not been previously possible. A Linux-based deployment of PD-ORAM is used here to design and build *privatefs*, a fully-functional oblivious network file system in which files can be accessed on a remote server with computational access privacy and data confidentiality.

An initial implementation was built on top of the Linux Network Block Device (NBD) driver, which is the simplest and most natural approach, since PD-ORAM already provides a block interface. However, NBD supports only serial, synchronous requests. To take advantage of the parallel nature of PD-ORAM, privatefs is instead built on FUSE.

A second attempt used *ext2fuse*, a FUSE-based ext2 im-

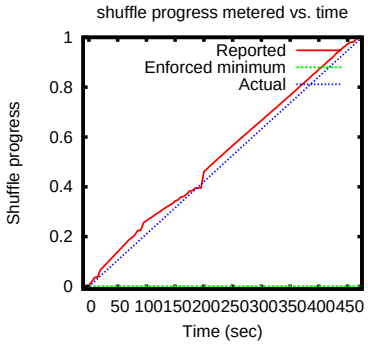

Figure 9: Level shuffle progress over time: The linear nature of the plot indicates the extrapolation based on partial shuffling is well done.

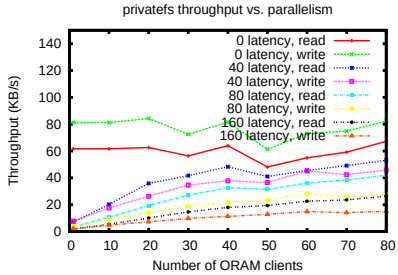

Figure 10: r/w performance vs. the number of ORAM Clients, for various network latencies. The zero latency environment is mostly unaffected by the degree of parallelism. For higher latencies, throughput grows with the number of parallel clients that offset the latency.

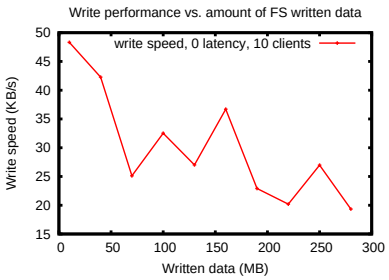

Figure 11: IOzone write performance vs. increasing file system written data. Write throughput is slowly decreasing, as expected due to the inherent slowdown in the (increasingly larger) ORAM.

plementation , by rerouting block access through PD-ORAM. However, thread safety difficulties prevented us from modifying it to support parallel writes or reads. Additionally, because of its nature as a block device file system, *ext2fuse* requires mechanisms for allocating blocks for files, such as block groups, free block bitmaps and indirect file block pointers inside inodes. These mechanisms are not all thread-safe and pose a challenge to synchronize. Moreover, locking the code using synchronization primitives would not result in a sufficient degree of parallelization.

Instead, we implemented our own *privatefs* using the C++ FUSE libraries. It fully leverages PD-ORAM parallelism and also takes advantage of the non-contiguous block labeling in a way that block-device file systems cannot.

Following the Linux file system model, in *privatefs* files are represented by inodes. Directories are inodes containing a list of directory entries; each directory entry is the name of a file or subdirectory along with its inode number. Inodes are numbered using 256-bit values and are mapped directly to ORAM blocks, such that inode x is stored in ORAM block x. Inodes hold metadata such as type, size and permissions. Both *privatefs* and (this instance of) PD-ORAM use 256-bit block identifiers and 4096-byte blocks.

Because the ORAM provides random access to 256-bit addressable blocks, a block can be allocated simply by generating a random 256-bit number. We take advantage of this in two ways. First, to read or write the i^{th} block of file with inode number x, the pair (x, i) is hashed with the collision-resistant SHA256 hash, yielding the 256-bit ORAM block ID for that file block. Second, when a new file is created, a 256-bit inode number is randomly generated, as opposed to maintaining and synchronizing access to an inode counter.

Our design eliminates the complexity of contiguous block device file systems and minimizes the need for locking when writing or reading files. As opposed to *ext2fuse*, *privatefs* does not incur the overhead of maintaining free block or inode bitmaps, grouping blocks into block groups, or traversing indirect block pointers to read files. The potential drawback is that sequential blocks of a given file will not be stored contiguously in the file system. However, this is harmless when using an ORAM, since there is no notion of sequential block numbers (which would compromise access privacy).

privatefs employs exclusive locks when reading and writing directories. In addition, an LRU cache is implemented to quickly retrieve an inode's data given its inode number and also for file path to inode number translation, which helps avoid long directory traversals (and associated locking). *privatefs* communicates with the ORAM server by means of a proxy (written in Java), which receives block requests from the file system and satisfies them using parallel connections to the ORAM server. This design choice affords us a higher degree of modularity, enabling us to connect *privatefs* to other ORAM schemes in the future.

We benchmarked privatefs along with our previous file system attempts, using a parallel workload writing ten, 512KB files simultaneously to the file system, then checking their integrity (also done simultaneously) using the `sha256sum` utility. For the single-client NBD implementation, throughputs ranged between 10.45KB/s and 14.80KB/s. For ext2fuse, throughputs ranged between 7.11KB/s and 9.62KB/s. In contrast, the performance results for the full implementation of *privatefs* discussed below indicate a major improvement of an order of magnitude.

Our benchmarks (Figure 10) indicate that *privatefs* is benefiting from the high degree of parallelism in high-latency environments. We ran five trials for each point, varying the number of ORAM clients used by the proxy to satisfy file system block requests and performance increased proportionally with the degree of parallelism. The average over the five trials is plotted.

Reads are less expensive than writes in the 40ms and 160ms trials, because the individual writes are performed synchronously, while the reads can be parallel, resulting in a higher overall degree of parallelism. On the other hand, the low latency in the 0ms trial prevents this parallelism from having an impact. The reads are, in turn, more expensive, due to inefficiencies in the de-amortization method resulting in slightly slower queries at later points in the process.

We also analyzed the behavior of increasing file system size (Figure 11) and used the widely used IOzone benchmark to test the write throughput as we wrote more data. Starting with an empty file-system we repeatedly ran a parallel write throughput test using IOzone which writes ten 1MB files and then rewrites them.

We compared *privatefs* to NFSv3 and ext4 using the IOzone workload. The *privatefs* tests wrote ten, 1MB files concurrently, while the NFS and ext4 tests wrote ten 1GB files concurrently. We decided to use larger file sizes when performing the tests for NFS and ext4 in order to minimize the impact of caching. The expected significantly

higher throughputs hovered around 57MB/s for NFSv3 and 138MB/s for ext4.

Thus, in this simple setup, *privatefs* features a modest throughput when compared to unsecured file systems. This is the inherent cost of achieving privacy. However, a few notes are in order to outline how performance can be scaled with increasing resources thrown at the problem.

SSDs. Deploying (multiple, Sections 2.3, 6.3) zero-latency media server-side would significantly impact performance. This is straightforward and relatively uninteresting research-wise. For example, deploying SSDs would shift bottlenecks and immediately increase throughput by an order of magnitude or more.

Block Sizes. privatefs has been benchmarked with 4KB blocks. One can straightforwardly increase their size by considering larger blocks and obtain an (artifically inflated) "higher throughput" up to the maximum available bandwidth. E.g., going from 4KB to 64KB blocks could increase "throughput" by another order of magnitude or more at the expense of wasted bandwidth. Similarly, we felt this is not interesting research-wise and should be decided on an application-specific basis.

Compute Power. Finally, to eliminate bottlenecks, large amounts of server-side resources can be thrown at the problem to speed up things even further. We posit that this is also not interesting – what ultimately counts is the usable bang for the buck achieved, i.e., in this case the throughput (at some fixed block-size) achieved per compute cycle spent server-side. Otherwise, results become meaningless – e.g., if instead of the test setup above we were to deploy a large number of compute cores, performance would increase almost linearly up to network saturation. With careful fine-tuning throughputs of hundreds of Mbps can be achieved without any changes in the base protocol. However, from a security point of view or research-wise in general this is not interesting but should be pursued in industrial R&D.

8. CONCLUSION

This paper includes mechanisms for secure parallel querying of existing ORAMs to increase throughput, a generalization of ORAM de-amortization, and implementation of an efficient ORAM based on these techniques, performing a transaction per second on a terabyte database in an average-latency network (a first). An implementation of *privatefs*, the first oblivious networked file system, is provided.

9. ACKNOWLEDGMENTS

Supported in part by NSF under awards 1161541, 0937833, 0845192, 0803197, 0708025. We would also like to thank the reviewers and our shepherd, Alina Oprea.

10. REFERENCES

[1] Boneh, D., Maziéres, D., and Popa, R. A. Remote oblivious storage: Making Oblivious RAM practical. Tech. rep., MIT, 2011. MIT-CSAIL-TR-2011-018 March 30, 2011.

[2] Goldreich, O., and Ostrovsky, R. Software protection and simulation on Oblivious RAMs. *Journal of the ACM 45* (May 1996), 431–473.

[3] Goodrich, M., and Mitzenmacher, M. MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. In *ICALP* (2011).

[4] Goodrich, M., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Oblivious RAM Simulation with Efficient Worst-Case Access Overhead. In *ACM Cloud Computing Security Workshop (CCSW)* (2011).

[5] Goodrich, M. T. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA* (2010).

[6] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Oblivious storage with low I/O overhead. *CoRR abs/1110.1851* (2011).

[7] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA* (2012).

[8] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA* (2012), Y. Rabani, Ed., SIAM, pp. 143–156.

[9] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 9–9.

[10] Olumofin, F., and Goldberg, I. Revisiting the computational practicality of private information retrieval. In *In Financial Cryptography and Data Security '11* (2011).

[11] Ostrovsky, R., and Shoup, V. Private information storage (extend abstract). In *IN PROCEEDINGS OF STOC* (1997), ACM Press, pp. 294–303.

[12] Pagh, R., and Rodler, F. F. Cuckoo hashing. *J. Algorithms 51* (May 2004), 122–144.

[13] Pinkas, B., and Reinman, T. Oblivious RAM revisited. In *CRYPTO* (2010), pp. 502–519.

[14] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious RAM with o((logn)3) worst-case cost. In *ASIACRYPT* (2011), pp. 197–214.

[15] Sion, R., and Carbunar, B. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium* (2007).

[16] Stefanov, E., Shi, E., and Song, D. Towards Practical Oblivious RAM. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium* (2012).

[17] Wang, S., Ding, X., Deng, R. H., and Bao, F. Private information retrieval using trusted hardware. In *Proceedings of the European Symposium on Research in Computer Security ESORICS* (2006), pp. 49–64.

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

[19] Williams, P., Sion, R., and Carbunar, B. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security* (2008), pp. 139–148.

[20] Williams, P., Sion, R., and Sotakova, M. Practical oblivious outsourced storage. *ACM Trans. Inf. Syst. Secur. 14* (September 2011), 20:1–20:28.