Combining ORAM with PIR to Minimize Bandwidth Costs

Jonathan Dautrich
Google, Inc.
Irvine, California
jjldj@google.com

Chinya Ravishankar
Computer Science and Engineering
University of California, Riverside
ravi@cs.ucr.edu

ABSTRACT

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns limit its reach. Even if the stored data are encrypted, access patterns may leak valuable information. Oblivious RAM (ORAM) protocols guarantee full access pattern privacy, but even the most efficient ORAMs proposed to date incur large bandwidth costs.

We combine Private Information Retrieval (PIR) techniques with the most bandwidth-efficient existing ORAM scheme known to date (ObliviStore), to create OS+PIR, a new ORAM with bandwidth costs only half those of ObliviStore. For data block counts ranging from 2^{20} to 2^{30}, OS+PIR achieves a total bandwidth cost of only 11X–13X blocks transferred per client block read+write, down from ObliviStore's 18X-26X. OS+PIR introduces several enhancements in addition to PIR in order to achieve its lower costs, including mechanisms for eliminating unused dummy blocks.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*security, integrity, and protection*

Keywords

Data privacy; Oblivious RAM; private information retrieval

1. INTRODUCTION

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns limit its reach. Even if data blocks are encrypted by the client before being stored on the server, data access patterns may still leak valuable information [5, 11, 12]. *Private Information Retrieval* (PIR) [3] and *Oblivious RAM* (ORAM) [9] techniques both offer provable access pattern privacy for outsourced data, each with their own advantages and disadvantages. In this work, we combine the strengths of existing ORAM and PIR constructs to create a new ORAM with reduced bandwidth costs.

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.

ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699117.

In PIR, data on the server may be encrypted or unencrypted, and the client issues an encrypted query for a particular bit or block of B bits. The server evaluates each query *homomorphically* (without decryption), returning the desired block without learning which block was requested. In order to achieve this degree of security, the server must evaluate each query over all bits in the database, making PIR computationally prohibitive for most applications [16].

In ORAM, blocks of data are always encrypted by the client before being stored on the server. Informally, the ORAM defines a protocol that dictates how the client should fetch, permute (shuffle), re-encrypt, and store blocks from and to the server in order to prevent the server from learning any information about the pattern of plaintext block requests. The ORAM protocol guarantees that for any two same-length sequences of plaintext block requests, the resulting patterns of encrypted block accesses are computationally indistinguishable to all observers other than the client. ORAM requires negligible computation, but may incur substantial bandwidth or storage overheads.

Definition 1. The *bandwidth cost* W of an ORAM or PIR technique is the number of blocks transferred for every block requested. Equivalently, it is the total number of bits transferred in order to retrieve B bits, where B is the block size. If we upload and download 3 blocks for each request, $W = 6$X.

Bandwidth cost is particularly important when applying ORAM to mobile devices, where bandwidth costs are substantial. Existing ORAMs have bandwidth costs polylogarithmic in the total number of blocks N [7, 9, 10, 13, 18, 20, 23]. The ObliviStore (OS) ORAM [18] has the lowest bandwidth cost of any single-server ORAM proposed to date. OS's bandwidth cost is roughly $(\log_2 N)$X, with no hidden constants, though it requires extensive client storage. Other ORAMs require less storage [13, 14, 20] or reduce response times [2, 6, 22], but all incur higher total bandwidth costs.

1.1 Our Proposal: OS+PIR

In this work, we show how to drastically reduce ObliviStore's bandwidth costs by combining it with PIR. The combination yields a new ORAM that we call OS+PIR that offers reduced bandwidth costs and permits tradeoffs between bandwidth cost, client/server storage, and computation. Combining ORAM and PIR was proposed in [14], but their scheme uses a different ORAM [15] and emphasizes constant client storage instead of low bandwidth costs.

OS+PIR treats its PIR component largely as a black box, so any efficient PIR technique may be used. Section 4 discusses how we use the Trostle-Parrish PIR [21], also used in [14], to reduce online bandwidth costs in OS+PIR. In OS, for each request, the client retrieves and decrypts one block from each of $O(\log N)$ levels. At most one of the decrypted blocks is *real* and all others are *dummies* (Section 3.2) that can be discarded. We use PIR to retrieve only the real block without revealing which block was accessed. Since we use PIR for only a small number ($O(\log N)$) of blocks, it is computationally feasible.

We divide OS+PIR's total bandwidth cost W into an ORAM component W_O and a PIR component W_P, where $W = W_O + W_P$. W_O depends on the ORAM's behavior, and W_P depends on the specific PIR and parameters such as block size. Applying PIR reduces W_O by roughly 30%.

In Section 5 we show how to amplify this reduction by altering the number and relative sizes of partition levels, balancing the reduced bandwidth cost with the resulting increases in PIR computation and server storage.

We present experimental results in Section 6. For systems with 2^{20} to 2^{30} blocks of 2MiB each, OS+PIR reduces the total bandwidth cost from OS's 18X–26X to only 11X–13X.

In the full version of the paper, we present the previously undiscovered OS issue of *unused dummy blocks*, which is exacerbated in OS+PIR by the increased level sizes. Unused dummies occur because ObliviStore makes more evictions than requests, which causes unnecessary block downloads. We show how to mitigate this issue by securely altering OS+PIR's eviction pattern and creating fewer dummy blocks. We also analyze two previously proposed techniques that use available client space to reduce bandwidth cost, reducing the eviction rate [18] and applying level caching [6], and show how to strike a reasonable balance between them.

2. RELATED WORK

Oblivious RAM was first proposed in [9] and required a bandwidth cost of $O(\log^3 N)$X blocks transferred per block requested, where N is the number of data blocks in the ORAM. Subsequent works have reduced the bandwidth cost to $O(\log^2 N/\log\log N)$X using constant client-side storage [13]. While constant client storage is desirable, it is not always necessary in practical settings. Recent works have reduced bandwidth costs to $O(\log N)$X by allowing additional client storage, specifically: $O(BN^v)$ client storage for constant $v > 0$ in [10], $O(B\log N)$ for large $B \in O(log^2 N)$ in [20], and $O(N\log N + B\sqrt{N})$ in [6, 18, 19].

Not all ORAMs that achieve $O(\log N)$X bandwidth cost are equally practical. For block sizes on the order of kilobytes or larger, the ObliviStore (OS) family of ORAMs [6, 18, 19] has the lowest practical bandwidth cost of any single-server ORAM proposed to date. OS's bandwidth cost is roughly $(\log_2 N)$X, with no hidden constants, though it requires extensive client storage. In contrast, Path ORAM [20] requires closer to $(8\log_2 N)$X, and more if client space is reduced using recursion. The scheme in [10] requires roughly $\log_2 N$ round-trips per request, but each trip may include several block transfers, making the total bandwidth cost at least 3 to 4 times that of OS. Since OS has the lowest bandwidth cost, we compare with it when evaluating OS+PIR.

Prior work [14] combined PIR with the tree-based ORAM of [15]. While their construction achieves the desirable property of constant client memory, it still requires $O(\log^2 N)$X bandwidth cost. In contrast, OS+PIR combines PIR with the partition-based OS [18] to achieve under $(1/2)(\log_2 N)$X bandwidth cost in practice. Multi-cloud oblivious storage [17] achieves very low bandwidth cost (under 3X), but makes the strong assumption of multiple non-colluding servers.

Table 1: Notation

N	Total number of real (data) blocks in an ORAM
B	Size of each data block (in bits)
b	A specific plaintext data block
W	Total bandwidth cost
W_O	ORAM component of total bandwidth cost W
W_P	PIR component of total bandwidth cost W
ϵ	Eviction rate
p	A partition used in ObliviStore or OS+PIR
p_r	Request partition
p_a	Assignment partition
p_e	Eviction partition
k	Level size factor
K	Level configuration consisting of level size factors
r	Number of real blocks in a sub-level
L_M	Total number of main levels in a single partition
L_S	Total number of sub-levels in a single partition
D_i	Maximum number of dummies in main level i
s	Number of noise bits in PIR

3. PRELIMINARIES

Key notation used throughout the paper is in Table 1.

3.1 Private Information Retrieval (PIR)

Private Information Retrieval (PIR) was first proposed in [3], and allows a client to retrieve specific bits from a server without revealing any information to the server, or any other observer, about which bits were accessed.

PIR comes in two flavors: *computational* and *information theoretic*. Computational PIR schemes are based on a hardness assumption, such that to retrieve information about a query, the adversary would need to solve a problem that is considered intractable. Information-theoretic PIR guarantees that query information remains secret, regardless of the adversary's computational resources, but generally requires an assumption of non-colluding servers [3]. The non-colluding server assumption is inconsistent with untrusted servers, so we use computational PIR in our work.

Each PIR query is encrypted by the client, sent to the server, and evaluated *homomorphically* (under encryption) by the server. The server returns the requested bit/block of bits but learns neither the plaintext contents of the query nor which bits were returned. PIR must perform a computational operation over every bit in the PIR database for every query in order to achieve its strong privacy guarantees.

Instead of using PIR for the entire database, we use it to reduce the bandwidth cost required to retrieve one of a small subset of a client's encrypted blocks that would otherwise all be returned as part of an ORAM protocol. In this scenario, the PIR database is small and the block size is relatively large, so the substantial bandwidth cost reduction can outweigh the small increase in computation. We discuss our use of PIR in more detail in Section 4.

3.2 Oblivious RAM (ORAM)

Oblivious RAM (ORAM) was first proposed in [9]. Like PIR, ORAM may be used to retrieve encrypted data from a server without revealing which data were accessed. ORAMs generally allow writes, while standard PIR is read-only.

Instead of evaluating queries homomorphically, ORAM defines a protocol for transferring and modifying encrypted blocks such that the underlying plaintext blocks accessed by different queries are unlinkable. ORAMs download encrypted blocks from the server to a trusted local client space, decrypt the desired data, then re-encrypt the blocks using a semantically secure encryption scheme to break correlations with previous encrypted *contents*. The ORAM then randomly permutes the blocks to break correlations with the previous block *position*, a process referred to as oblivious shuffling. Some ORAMs also use *dummy blocks*, which are indistinguishable from real blocks but contain irrelevant data, and download extra blocks, to break correlations.

ORAM security is defined as follows: For any two sequences of block requests of the same length, the resulting patterns of encrypted block accesses must be computationally indistinguishable to all observers other than the client [19]. Equivalently, the output of a simulator that has no access to any of the secret information (block contents and requested block addresses) should be able to produce a sequence of encrypted block transfers that is indistinguishable from that of the actual ORAM [13].

3.3 ObliviStore (OS)

OS+PIR builds on the ObliviStore (OS) ORAM [18]. A full description of OS and its underlying ORAM [19] is too extensive to include here, but we review the aspects most relevant to OS+PIR. For N blocks of B bits each, OS uses a relatively large amount of client storage, $O(B\sqrt{N}+N\log N)$ bits with small constants, in order to achieve a low bandwidth cost of $(\log_2 N)$X.

3.3.1 Partition and Level Structure

In OS, blocks stored on the server are arranged logically into $O(\sqrt{N})$ *partitions*, each of which contains $O(\sqrt{N})$ blocks. Each partition is a simplified hierarchical ORAM, similar to those of [9], with roughly $\log_2 \sqrt{N}$ levels. The lowest level in each partition (level 0) holds 1 real and 1 dummy block. Successive levels double in size, so level i has real-block capacity $r_i = 2^i$ and starts with 2^i dummy blocks. At any given time, each level may be occupied or empty, and only half the levels are occupied on average.

3.3.2 Requests and Evictions

Each block request involves three steps:

1) Partition Request When the client issues a request for block b, OS directs the request to the partition p_r containing b. The choice of p_r is deterministic, but appears random to an observer since b was previously assigned to a randomly chosen partition. OS then downloads exactly one block from every non-empty level in p. OS fetches b from whichever level contains it, and fetches a dummy block from every other level. Since levels were previously randomized, each fetched block appears randomly chosen from its level. After downloading the blocks, OS discards all dummies, returns b to the client then *assigns* b to a new partition.

2) Assignment: After reading and optionally updating b, OS encrypts it and assigns it to a partition p_a chosen uniformly at random. Each p_a maintains a local, hidden *eviction queue* of blocks assigned to p_a but not yet evicted to the server. OS assigns b to p_a by adding it to the end of p_a's eviction queue, but does not immediately evict it.

3) Eviction: After assigning b to p_a, OS independently chooses at least one partition p_e and evicts either the next block from p_e's eviction queue, or a dummy block if the queue is empty. We perform ϵ evictions after each request, where ϵ is the real-valued *eviction rate*.[1] If ϵ exceeds 1.0, OS adds the fractional component $\epsilon - 1.0$ into a global accumulator. When the accumulator reaches 1.0, OS makes another eviction and decrements the accumulator. Eviction partitions (p_e) may be chosen deterministically or randomly, as long as the choice is independent of p_a [19].

Choosing p_a randomly guarantees that future requests for b will appear to access a random partition. Choosing p_e independently of p_a prevents an observer from learning p_e and thus from tracking b between partitions. Thus, the independence of p_a and p_e is critical to OS's security.

Since p_a and p_e are chosen independently, blocks may accumulate in eviction queues. OS calls the space used by these assigned but not yet evicted blocks the *eviction cache*. Revealing the eviction cache size may leak information about prior choices of p_a, so a fixed amount of space sufficient for the eviction cache is reserved up-front. Statistically, the higher ϵ, the less space is required for the eviction cache.

3.3.3 Shuffling

Let level i in partition p initially contain r_i real and r_i dummy blocks. Once r_i evictions have been made to p since i was created, i is scheduled to be *re-shuffled*. Shuffling i consists of:

1. Downloading all blocks left in level i
2. Removing remaining dummies
3. Inserting any evicted blocks
4. Generating any additional dummies
5. Randomly permuting and re-encrypting all blocks
6. Uploading all blocks to a new level with $2r_i$ real and $2r_i$ dummy blocks

When two levels i and $i-1$ are both ready to be re-shuffled, the shuffle *cascades* upward. Level sizes increase by factors of 2, so every time level i is ready to be re-shuffled, all lower levels are also necessarily ready. When shuffling all levels up to i, we download all remaining blocks from levels 0–i and upload blocks to level $i+1$ with $2r_i$ real blocks, leaving levels 0–i empty to accommodate future evictions (see Figure 1).

3.3.4 Early Shuffle Reads

If level i has more than half its original $2r_i$ blocks remaining, then there is at least one dummy block in i to return. If instead i has at most r_i blocks remaining, it is possible that all the blocks are real, so to maintain obliviousness OS must *treat* such blocks as real. We call such blocks *early shuffle reads*, since they would eventually be downloaded as part of the upcoming shuffle. OS refers to such blocks as either *early cache-ins* or *real cache-ins* depending on the context.

Early shuffle reads are relatively rare, and are caused by delayed shuffles in OS. Since OS performs a constant amount of work per request, some levels to shuffle may have not yet been downloaded when a later request arrives, causing the

[1] OS uses slightly different notation, where v is the *background eviction rate*, equivalent to $1 - \epsilon$.

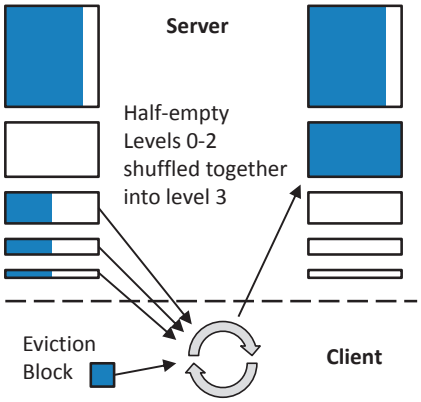

Figure 1: When shuffles cascade upward in OS, all levels to be re-shuffled are downloaded, shuffled with eviction blocks, and uploaded to a higher level.

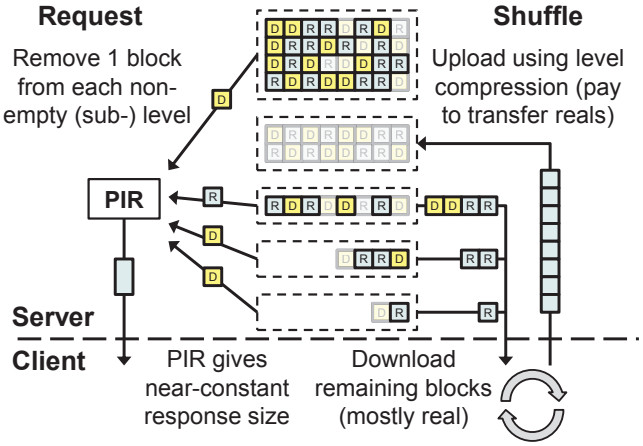

Figure 2: In OS+PIR, PIR reduces the response cost for each request to near-constant. During shuffling, only blocks remaining in each level (largely real) need be downloaded. With level compression, we need only transfer data of size equal to the uploaded reals. In all, dummy transfers are "free" — we effectively pay only to transfer real blocks.

early shuffle read. Other than early shuffle reads, all but one of the downloaded blocks are guaranteed to be dummies. Early shuffle reads are downloaded separately and stored.

3.3.5 *Level Compression*

OS's level compression algorithm [19] lets the client send k real and k dummy blocks to the server using only kB bits. The technique uses a pre-shared Vandermonde matrix $M_{2k\times k}$ to encode the k real blocks and their positions into a "compressed" kB-bit stream. The server decompresses the stream to get $2k$ blocks: k real and k dummies containing random data derived from the decompression. The dummy and real blocks are indistinguishable and intermixed.

We can alter the number of dummies generated by changing the number of rows in M from $2k$ to the desired total block count. This flexibility becomes important in Section 5 where we use it to reduce OS+PIR's bandwidth cost.

3.3.6 *Bandwidth Costs*

In OS, the client has enough space to store all $\sqrt{N}$ blocks from any given partition, and to store the location of all N blocks. Since every partition fits entirely in client memory, re-shuffling requires that each block be downloaded and uploaded only once. (Less client space we necessitate an expensive oblivious shuffling algorithm.) Thus, to shuffle r real blocks, we need only transfer $3r$ blocks: r real downloads, r dummy downloads, and r uploads for level compression.

The total bandwidth cost W of OS is determined by the number of times each block must be re-shuffled per request. Each partition has roughly $(\log_2\sqrt{N})/2 = (\log_2 N)/4$ occupied levels at any time. Each of the $\sqrt{N}$ real blocks is shuffled once per occupied level per $\sqrt{N}$ evictions, for a total $(3/4)\log_2 N$ block transfers per eviction. Since ϵ is the number of evictions per request, we get an expected OS cost:

$$W \approx \frac{3\epsilon}{4}\log_2 N \qquad (1)$$

OS reports an actual cost $W \approx \log_2 N$ for $\epsilon = 1.3$ [18]. The slight discrepancy with Equation 1 can be accounted for by the problem of *unused dummies*, addressed in [4].

4. INTEGRATING PIR

In ObliviStore (OS), each request fetches one block from each level in a partition. At most one fetched block is real and all others are dummies, except in the case of *early shuffle reads*, which are returned individually. Since returned dummies are discarded, they are only transferred to mask the real block's identity. We want to retrieve the real block and hide its identity without paying to transfer dummies.

The recently-proposed Burst ORAM [6] combines these fetched blocks using XOR and returns a single block. The client reconstructs dummy blocks locally and subtracts them out of the combined block to recover the real. The XOR optimization is incompatible with OS's level compression, since level compression constructs dummies from reals during decompression, but XOR requires that dummies be generated by the client [6]. Thus, Burst ORAM avoids paying to *download* dummies, but overall savings are negated by the lack of level compression, which avoids paying to *upload* dummies.

In OS+PIR, we instead use PIR to retrieve the real block. Since PIR makes no stipulations on dummy block content, it can be used with level compression. PIR itself incurs a bandwidth cost W_P determined by block size B, number of blocks (levels) over which it queries, and the PIR scheme's properties. Other than W_P, dummy block transfers are essentially free (Figure 2), reducing W_O by roughly 30% up-front, and enabling additional cost-saving modifications (Section 5).

Given that the PIR's computational hardness assumption holds, the PIR operation leaks no information about which level's block was fetched. From a security standpoint, using PIR is therefore equivalent to OS's approach of downloading each level's block and discarding the dummies. Thus, OS+PIR's security guarantees are precisely those of OS except for the added PIR hardness assumption.

4.1 Choosing a PIR Technique

In OS+PIR, we execute the PIR protocol once per block request. Each PIR instance operates over a *PIR database*

of $L_S \in O(\log N)$ blocks, one per level, and returns a single block. Since we query over $O(\log N)$ blocks, PIR is far more computationally feasible than when used over all N blocks.

PIR schemes often measure bandwidth in terms of the cost of returning the full PIR database. To remain consistent with Definition 1, we instead measure PIR bandwidth cost as W_P: the total amount of data transferred during each PIR operation, divided by B. We want a PIR scheme with low W_P (near the optimal 1X). Since OS's bandwidth cost is already low, even a W_P of 10X could negate any advantages of using PIR. Ideally, W_P should be constant: independent of L_S and B. In practice, we can use any PIR that offers low, nearly-constant W_P for small L_S and large B.

One candidate is the Gentry-Ramzan PIR [8], which offers a constant W_P with a theoretic 2X minimum (closer to 4X in practice), but incurs substantial computation costs. Another option is the more computationally-efficient Trostle-Parrish PIR [21], which offers low, but not constant, W_P when $B \gg L_S$. OS+PIR treats its PIR as a black box, so any PIR that meets our criteria can be used. For now, we use the Trostle-Parrish PIR [21] used in [14], due to its simplicity and low bandwidth and computation complexity.

Due to space constraints, we defer a detailed discussion of the Trostle-Parrish PIR to the full version of our paper [4].

5. ALTERING LEVEL SIZE FACTORS

Combining PIR with OS's level compression technique effectively gives OS+PIR free dummy block downloads and uploads, aside from PIR computation and bandwidth (W_P) costs. As noted in Section 3.3.5, we can modify level compression to produce additional dummy blocks at no extra cost. Similarly, given near-constant W_P, we can query over any number of additional levels at no extra bandwidth cost. We now show how to use these properties to reduce the bandwidth cost of OS+PIR's by altering level sizes.

In OS, successive levels increase in size by a factor of 2, yielding $\log_2 \sqrt{N}$ levels per partition. In OS+PIR, we allow successive levels to increase by any integer factor. Let k_i be the *level size factor* of main level i, which defines the sub-level real-block capacity ratio r_i/r_{i-1}. We must allow up to $k_i - 1$ instances or *sub-levels* of main level i, which when shuffled together with all lower levels become a single sub-level of main level $i+1$. The real-block capacity of a sub-level in level i is given by $r_i = \prod_{j=0}^{i-1} k_j$.

To simplify the presentation of ideas throughout this Section, we assume that OS+PIR uses $\epsilon = 1.0$ (exactly one eviction per request). We address larger eviction rates in the full version of the paper [4].

5.1 Effects of Increasing Level Size Factors

We simplify our discussion of the high-level effects of increasing level size factors (k_i values), by assuming $k_i = k$ for all i, where $k = 2$ in OS. Figure 3 shows two level configurations ($k = 2$ and $k = 4$) for a partition with 15 real blocks. We discuss non-uniform level size factors and special handling of the top level later in this Section. Increasing k has the following effects:

It increases the total number of sub-levels L_S. For a partition of $\sqrt{N}$ blocks we need $L_M = \log_k \sqrt{N}$ main levels. With $k-1$ sub-levels per main level, the total number of sub-levels is given by $L_S = (k-1)\log_k \sqrt{N}$, which increases almost linearly with k. To maintain obliviousness, we must

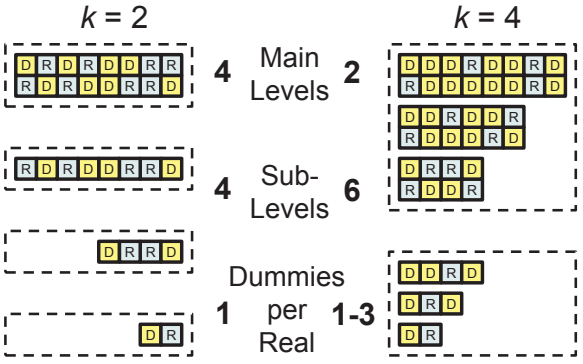

Figure 3: Level configurations with size factors $k = 2$ and $k = 4$, both with 15 real-block capacity. When shuffling, all sub-levels in a main level combine to form one new sub-level in next largest main level. $k = 4$ configuration has fewer main levels, thus lower shuffling costs. $k = 2$ has fewer dummies and sub-levels, thus lower disk and PIR costs.

fetch one block from every sub-level during each request. Thus, without PIR, the bandwidth cost W would increase almost linearly with k, which is why ObliviStore uses only $k = 2$. With near-constant W_P, the effect on W is negligible. Even with PIR, the number of blocks that must be read from disk to satisfy a request increases with k.

It increases level and sub-level lifetimes. The ith main level holds $k-1$ sub-levels containing k^i real blocks each, and is re-shuffled after every k^{i+1} requests. Once the ith level is shuffled into a higher level, it stays empty for k^i requests before its first new sub-level is created. The first sub-level must live through $(k-1)k^i$ requests and still have k^i blocks left over to avoid early cache-ins. Thus we need $(k-1)k^i$ dummies in addition to the k^i reals, for a total of k^{i+1} blocks. The second sub-level lives for k^i fewer requests, so it needs only $(k-2)k^i$ dummies, and so on. The increased number of dummies also increases server storage to a factor of roughly k, which we address in Section 5.2.

It decreases the ORAM component bandwidth cost W_O. Between every shuffle (k^{i+1} requests), the ith level receives $k^{i+1}(k-1)/2$ dummy blocks and $(k-1)k^i$ real blocks in all. The number of shuffle *downloads* per request is only $((k-1)/k)\log_k \sqrt{N}$, since we only pay to download the remaining $(k-1)k^i$ blocks from each sub-level. With level compression, the number of shuffle *uploads* per request is the same, giving:

$$W_O \approx 2\frac{k-1}{k}(\log_k \sqrt{N})\text{X} \quad (2)$$

Thus, for OS+PIR with $N = 2^{32}$ real blocks and $\epsilon = 1.0$, increasing k from 2 to either 4, 16, or 64 should reduce W_O from roughly 16X to either 12X, 7.5X, or 5.25X, resp.

5.2 Non-Uniform Level Size Factors

A major limitation of using a large fixed level size factor k is that it increases server storage cost. The ith level stores a total of k^{i+1} blocks, only k^i of which are real, for a server storage factor of roughly k. However, the bulk of the extra dummy blocks, at least $k-1$ of each k, are stored in the largest level. With varying k_i, level $i \geq 1$ stores D_i dummy

blocks in the worst case, given by:

$$D_i \approx \left(\frac{k_i}{2}\right)^2 \cdot \prod_{j=0}^{i-1} k_j \quad (3)$$

By allowing level size factors to differ, specifically using smaller k_i for larger levels and vice-versa, we can mitigate the storage cost increase but keep the number of levels small.

For example, consider the two configurations $K_1 : (k_0 = 2^5, k_1 = 2^5, k_2 = 2^5)$ and $K_2 : (k_0 = 2^7, k_1 = 2^5, k_2 = 2^3)$. For both, the real-block capacity is roughly $\prod_{j=0}^{2} k_j = 2^{15}$. However, for K_1 we get $D_0 + D_1 + D_2 = 2^8 + 2^{13} + 2^{18} \approx 2^{18}$ dummy blocks total, while K_2 gives $D_0 + D_1 + D_2 = 2^{12} + 2^{15} + 2^{16} \approx 2^{16}$ dummies total, reducing the server storage factor from 8X to 2X.

By increasing level factors doubly-exponentially for a partition with $\sqrt{N}$ blocks, we can asymptotically reduce the main level count to $L_M \approx \log_2 \log_2 \sqrt{N}$, incurring only a L_M server storage cost factor. Consider the configuration:

$$K : (k_0 = 2^{(\log_2 \sqrt{N})/2}, k_1 = 2^{(\log_2 \sqrt{N})/4},$$
$$k_3 = 2^{(\log_2 \sqrt{N})/8}, \ \ldots, \ k_{L_M-3} = 2^8$$
$$k_{L_M-2} = 2^4, k_{L_M-1} = 2^2, k_{L_M} = 2^1).$$

Since the level factor exponents in K grow exponentially from 1 to $(\log_2 \sqrt{N})/2$, we have $L_M \approx \log_2 \log_2 \sqrt{N}$, and thus $W_O \approx (2 \log_2 \log_2 \sqrt{N})$X. Applying Equation 3, we see that $D_i \approx \sqrt{N}$ for all i. Since the real-block capacity is $\sqrt{N}$, the server storage overhead is just $(\log_2 \log_2 \sqrt{N})$X.

5.2.1 Practical Limits on Level Size Factor Growth

Unfortunately, increasing or skewing level size factors also increases the maximum total sub-level count L_S given by:

$$L_S \leq \sum_i (k_i - 1). \quad (4)$$

For every request, we must fetch L_S blocks from disk and perform PIR over L_S blocks. Thus, disk read and PIR computation costs are at least proportional to the largest k_i, limiting growth in practice.

In the simple example above, K_2 suffers from $L_S \approx 165$, while K_1 has only $L_S \approx 93$. For such small skews, the difference is not dramatic, but in a comparable configuration with all $k_i = 2$, we have only $L_S \approx 15$. In the double-exponential growth example, $k_0 = 2^{(\log_2 \sqrt{N}/2)} = N^{1/4}$. For large databases with $N \geq 2^{32}$, we end up with $L_S \geq k_0 \geq 2^8$. Such large L_S values could easily make disk and PIR costs outweigh any benefit of reduced bandwidth cost in practice.

A more practical approach is to follow double-exponential growth only up to a maximum level factor determined by a fixed acceptable value of L_S that can be accommodated by the disk array and PIR computation hardware. In Section 6.2 we empirically evaluate the impact of different level size configurations on L_S.

6. EXPERIMENTS

We ran simulations to compare OS+PIR with OS [18]. OS+PIR is equipped with the additional bandwidth-saving enhancements discussed in the full version of the paper [4]: eliminating unused dummies, evicting to request partition, caching smallest levels, and shuffling largest jobs first. For fairness, we also compare with a modified OS equipped with these same enhancements (but not PIR).

In all our experiments, we assume a 2MiB block size ($B = 2^{24}$ bits). The large B is needed primarily to keep PIR bandwidth cost W_P low when using the Trostle-Parrish PIR. Changing B has little effect on the ORAM component bandwidth cost W_O. For the unmodified OS, we use $\epsilon = 1.3$ as in [18], and use $\epsilon = 1.1$ for the modified OS and for OS+PIR.

The full version [4] includes additional experiments evaluating the individual effects of each bandwidth enhancement and exploring the effects of changing eviction rates.

Table 2: Effects of changing the level configuration K for $N = 2^{28}$ block count, 512TiB capacity, 512GiB total client storage. Product of all level size factors for each K is 2^{16}.

K	Server Storage Factor	W_O	Avg L_S	Max L_S	W_P
$(2, \ldots, 2)$	2.9	16.1X	6.7	14	2.7X
$(4, \ldots, 4, 2, 2)$	3.1	13.1X	9.5	20	2.8X
$(4, 8, 8, 8, 8, 2, 2)$	3.5	11.2X	14.5	30	3.0X
$(4, 16, 8, 8, 4, 2, 2)$	3.2	11.3X	16.5	34	3.0X
$(4, 16, 16, 16, 2, 2)$	4.3	9.9X	23.0	47	3.2X
$(4, 32, 16, 16, 2)$	5.8	9.1X	30.5	62	3.4X
$(4, 128, 16, 4, 2)$	4.3	9.1X	72.5	146	4.1X
$(4, 128, 64, 2)$	15.7	7.6X	95.0	191	4.4X

6.1 ORAM Simulator

We evaluated bandwidth costs for OS+PIR and ObliviStore using a simulator written in Java. Since ORAM behavior is oblivious, performance is independent of the specific sequence of blocks requested. Thus, for efficiency, the simulator uses counters to represent the number of remaining blocks in each level of each partition, and avoids storing block IDs and contents explicitly. Since we are primarily interested in bandwidth and computation costs, the simulator does not explicitly measure costs of permuting blocks, looking up IDs, or performing disk reads. Block encryption costs are also not logged, as they are dominated by PIR costs.

Each experiment includes a run-up and evaluation phase of $4N$ requests each. We count the total number of blocks transferred (uploads plus downloads) during the evaluation phase, and divide by $4N$ to get W_O. For OS+PIR we also record the number of sub-levels L_S accessed during each request. L_S varies across requests depending on the number of empty sub-levels in each partition.

6.2 Varying Level Size Configuration K

Table 2 shows the results of running $OS+PIR$ for various K given a fixed $N = 2^{28}$ and 512GiB of client storage. We use the same maximum real-block partition capacity for all K, so the product of all level factors in each K is fixed (2^{16}). As predicted in Section 5.2.1, using larger level factors, and thus fewer main levels $L_M = |K|$, greatly reduces ORAM bandwidth cost W_O, but also increases L_S, which in turn increases PIR and disk access costs. Server storage costs increase substantially when level factors for the highest levels are increased, even when bandwidth costs remain the same (see $K = (4, 32, 16, 16, 2)$ and $K = (4, 128, 16, 4, 2)$).

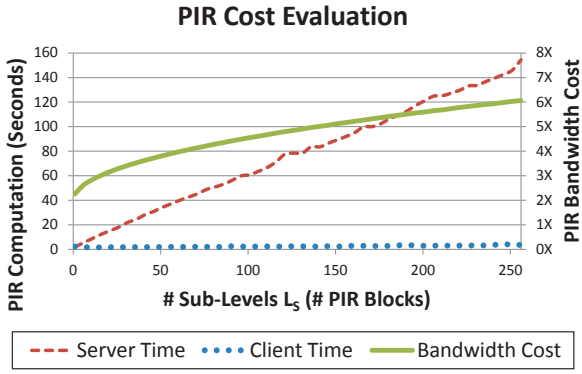

Figure 4: Timing and bandwidth costs of Trostle-Parrish PIR implementation for varying numbers of blocks in PIR database, using $s = 512$ bits noise and 2MiB block size.

6.3 PIR Implementation

We implemented the Trostle-Parrish PIR [21], as described in Section 4, using Java. Our implementation caches partial sums to avoid redundant computations, but is otherwise unoptimized. For all our experiments, we used $s = 2^9$ bits of extra noise in the PIR as discussed in the full version [4].

We measure wall-clock times running PIR on a single thread of a third generation Amazon Web Services (AWS) Elastic Compute Cloud [1] instance (half of a c3.large instance), equivalent to 3.5 AWS ECUs. As of May 2014, the cost for running the full c3.large instance was \$0.105/hour, giving an approximate PIR cost of $\$1.46 \times 10^{-5}$/second on a single thread. Figure 4 gives PIR time and bandwidth costs for L_S up to 2^8, with a maximum server time under 160s, equivalent to \$0.0023 per 2MiB block request.

We report times for a single thread to simplify cost analysis. However, server computation within each PIR operation can be trivially parallelized to at least $\sqrt{BL_S}$ threads. Since clients likely have less available parallelism, we expect the small reported client times, not the large server times, to correspond to real-world latencies.

6.4 Evaluating OS+PIR for Mobile Devices

We start by evaluating OS+PIR on parameters suitable to current mobile devices. We consider an OS+PIR with $N = 2^{22}$ of our 2MiB blocks, giving a server storage capacity of 8TiB. We allocate 64GiB total client storage, such that the ORAM increases effective storage capacity by a factor of 128. We can alter this factor by changing N (Section 6.5).

Table 3 shows our results. Comparing the modified and unmodified versions of OS, we see that our enhancements offer a slight improvement on their own, reducing total bandwidth cost W from 21.4X to 18.2X. Adding PIR offers another improvement, bringing W down to 16.2X. Finally, increasing the level factors reduces W to as little as 11.2X, but increases server storage and PIR computation costs.

We also give per-request costs in US cents (¢) for each scheme for two benchmark bandwidth costs. On one extreme we have the AWS [1] bandwidth cost of \$0.12 per GB for the first 10TiB per month, for a cost of 0.025¢ per 2MiB block. On the other we have cellular data, which may cost as much as \$10 per GB, for a cost of 2.10¢ per 2MiB block. At \$0.12/GB, OS+PIR roughly breaks even, with its added PIR computation cost canceling out the reduced bandwidth costs. At \$10/GB, OS+PIR is a clear win, as the bandwidth cost savings far outweigh the PIR cost, cutting total cost down to nearly half that of OS. OS+PIR is clearly most cost-effective when bandwidth costs dominate, as would be the case for mobile devices.

6.5 Varying Block Count N

Table 4 shows client/server space consumption and total bandwidth cost W as N increases. We show results for the unmodified ObliviStore, OS+PIR with $K = (2, \ldots, 2)$, and a custom K chosen to minimize L_M while keeping server storage costs low. Required client storage scales with $\sqrt{N}$, and for each N it is nearly the same for all three schemes. As N grows, the capacity/client space ratio grows as well, from 68 for $N = 2^{20}$ to 2189 for $N = 2^{30}$. For a large 2PiB database, the client needs 1TiB storage, and OS+PIR reduces W from 26.4X to 13.0X.

The bandwidth savings of OS+PIR depend on reducing the number of main levels $L_M = |K|$. However, to keep server storage costs low, we must use small level factors for the highest levels, limiting savings for small N. As N grows, our advantage increases, as L_M grows more slowly in OS+PIR than in OS due to OS+PIR's larger level factors.

7. CONCLUSION

We have presented OS+PIR, a new ORAM that combines the bandwidth-efficient ObliviStore (OS) ORAM [18] with PIR techniques to minimize total bandwidth costs. We have shown how to re-engineer OS to accommodate levels of varying relative sizes in order to fully exploit PIR, exposing a tradeoff between bandwidth cost, server computation, and server storage. OS+PIR also includes several enhancements that further reduce costs, including mechanisms for eliminating the unnecessary dummy blocks introduced in OS.

In all, OS+PIR achieves bandwidth costs at least 2 times lower than those of the already-efficient OS, making it especially advantageous for mobile devices, where bandwidth costs dominate. In other settings, OS+PIR's effectiveness is currently limited by its high PIR computation cost. However, since we can easily swap out the PIR protocol, OS+PIR can also be gainfully applied to less bandwidth-constrained settings as more efficient PIR schemes emerge.

8. ACKNOWLEDGEMENTS

This work was completed while the first author was a student at UC Riverside, and was supported by the National Physical Science Consortium Graduate Fellowship and by grant N00014-07-C-0311 from the Office of Naval Research.

9. REFERENCES

[1] Amazon web services. `http://aws.amazon.com`, May 2014.

[2] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.

[3] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

Table 3: Comparison of different bandwidth-efficient protocols given parameters tuned for current mobile devices with at least 64GiB storage. Common parameters: $N = 2^{22}$ data blocks, 2MiB block size, 64GiB total client storage. In practice, server latency will be much lower than reported single-thread times (Section 6.3).

				PIR Comp./Req.			Bandwidth Cost/Req.			Total Cost/Req.	
Protocol	ϵ	Level Factors	Server Storage Factor	Server Time (s)	Server Cost (¢)	Client Time (s)	PIR (W_P)	ORAM (W_O)	Total (W)	In ¢ at 12¢ / GB	In ¢ at $10 / GB
ObliviStore	1.3	$(2, \ldots, 2)$	3.2	—	—	—	—	21.4X	21.4X	0.054	44.88
ObliviStore Mod.	1.1	$(2, \ldots, 2)$	2.9	—	—	—	—	18.2X	18.2X	0.046	38.17
OS+PIR	1.1	$(2, \ldots, 2)$	2.9	6.04s	0.009	0.78s	2.6X	13.6X	16.2X	0.050	33.98
OS+PIR	1.1	$(4, 32, 16, 2, 2)$	4.5	20.55s	0.030	0.92s	3.2X	8.5X	11.7X	0.059	24.56
OS+PIR	1.1	$(4, 64, 16, 2)$	6.4	31.62s	0.046	0.99s	3.6X	7.6X	11.2X	0.074	23.53

Table 4: Effect of increasing N on client/server storage and bandwidth cost, with $\epsilon = 1.1$ for OS+PIR, $\epsilon = 1.3$ for ObliviStore.

		ObliviStore			OS+PIR $K = (2, \ldots, 2)$			OS+PIR Custom K			
N	Capacity (TiB)	Client (GiB)	Server (TiB)	W (X)	Client (GiB)	Server (TiB)	W (X)	K	Client (GiB)	Server (TiB)	W (X)
2^{20}	2	30	6.2	18.2	30	5.9	14.3	$(4, 64, 8, 2)$	31	11.2	10.9
2^{22}	8	60	24.8	19.8	61	23.4	15.3	$(4, 16, 16, 4, 2)$	62	31.6	11.4
2^{24}	32	119	99.2	21.4	122	93.3	16.5	$(4, 32, 16, 4, 2)$	123	127.6	12.0
2^{26}	128	239	396.2	23.1	244	373.0	17.6	$(4, 64, 16, 4, 2)$	245	522.4	12.5
2^{28}	512	478	1584.6	24.7	488	1491.4	18.7	$(4, 64, 32, 4, 2)$	491	2452.4	12.7
2^{30}	2048	958	6335.8	26.4	980	5962.9	19.8	$(4, 64, 32, 8, 2)$	990	9756.0	13.0

[4] J. Dautrich. *Achieving Practical Access Pattern Privacy in Data Outsourcing*. PhD thesis, University of California, Riverside, 2014.

[5] J. Dautrich and C. Ravishankar. Compromising privacy in precise query protocols. In *Proc. EDBT*, 2013.

[6] J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *Proc. USENIX Security*, 2014.

[7] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Proc. PETS*, 2013.

[8] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Automata, Languages and Programming*, pages 803–815. Springer, 2005.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.

[11] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. NDSS*, 2012.

[12] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on data and application security and privacy*, pages 235–246. ACM, 2014.

[13] 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 Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[14] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS*, 2014.

[15] E. Shi, H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.

[16] R. Sion. On the computational practicality of private information retrieval. In *Proc. NDSS*, 2007.

[17] E. Stefanov and E. Shi. Multi-Cloud Oblivious Storage. In *Proc. ACM CCS*, 2013.

[18] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. IEEE Symposium on Security and Privacy*, 2013.

[19] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. Proc. NDSS, 2012.

[20] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proc. ACM CCS*, 2013.

[21] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Information Security*, pages 114–128. Springer, 2011.

[22] P. Williams and R. Sion. Sr-oram: Single round-trip oblivious ram. *Proc. ACNS, industrial track*, pages 19–33, 2012.

[23] P. Williams, R. Sion, and A. Tomescu. PrivateFS: A parallel oblivious file system. In *Proc. ACM CCS*, 2012.