A

Access Privacy and Correctness on Untrusted Storage

Peter Williams, petertw@cs.stonybrook.edu, Stony Brook University
Radu Sion, sion@cs.stonybrook.edu, Stony Brook University

We introduce a new practical mechanism for remote data storage with *access pattern privacy* and *correctness*. A storage client can deploy this mechanism to issue encrypted reads, writes, and inserts to a potentially curious and malicious storage service provider, without revealing information or access patterns. The provider is unable to establish any correlation between successive accesses, or even to distinguish between a read and a write. Moreover, the client is provided with strong correctness assurances for its operations—illicit provider behavior does not go undetected. We describe a practical system that can execute an unprecedented *several queries per second on terabyte-plus databases* while maintaining *full computational privacy* and *correctness*.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Security, integrity, and protection

Additional Key Words and Phrases: Data Outsourcing, Private Information Retrieval, Access Privacy, Oblivious RAM

ACM Reference Format:
ACM Trans. Info. Syst. Sec. V, N, Article A (January YYYY), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

As networked storage architectures become prevalent—e.g., networked file systems and online relational databases in sensitive infrastructures such as email and storage portals, libraries, health and financial networks—protecting the confidentiality and integrity of stored data is paramount to ensure safe computing. Such data is often geographically distributed, stored on potentially vulnerable remote servers or transferred across untrusted networks; this adds security vulnerabilities compared to direct-access storage.

Moreover, today, the remote servers are increasingly maintained by third party storage vendors. Most third party storage vendors do not provide strong assurances of data confidentiality and integrity. For example, personal emails and confidential files are being stored on third party servers such as Gmail [Google 2012], Xdrive [Xdrive 2012], Amazon S3 [Amazon Web Services LLC 2012], and Ubuntu One [Canonical 2012].

Privacy guarantees of such services are at best declarative and often subject customers to unreasonable fine-print clauses—e.g., allowing the server operator (and thus malicious attackers gaining access to its systems) to use customer behavior for commercial profiling, or governmental surveillance purposes [CBS 2006].

A preliminary version of this paper was presented at CCS 2008 in [Williams et al. 2008].
The authors are supported in part by the NSF through awards 0937833, 0845192, and 0803197. The authors also wish to thank Motorola Labs, Microsoft Research, NGC, Nokia, IBM Research, the IBM Cryptography Group, CEWIT, and the Stony Brook Office of the Vice President for Research.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

To protect data stored in such an untrusted server model, security systems should offer assurances of data confidentiality and access privacy. As a first line of defense, to ensure confidentiality, all data and associated meta-data can be encrypted at the client side using non-malleable encryption, before being stored on the server. The data remains encrypted throughout its lifetime on the server and is decrypted by the client upon retrieval.

Encryption provides important privacy guarantees at low cost. Encryption, however, is only a first step, as significant information is still leaked through the access pattern of encrypted data. To consider a simple example, suppose an adversarial storage provider knows that a portion of the encrypted database corresponds to a sorted keyword index. The adversary can then correlate plaintext keywords, identified by their position in the index, to documents(!), by simply observing which locations in the encrypted index are updated when a new encrypted document is uploaded. This is an example of revealed access patterns leading to a direct violation of data privacy.

This example barely touches the surface of what can be revealed through access patterns. The memory locations accessed by an AES implementation, for example, indicate the S-boxes being used and the encryption key.[1] More serious implications arise from the property that the utility of any external information an adversary might have is multiplied by combining with access pattern information. Moreover, such leaks cannot be efficiently patched in the same way that other side channels, such as timing attacks, can be prevented. In general, it is difficult to bound the amount of information leaked by access patterns. Except in the most abstracted cases, data privacy cannot exist without access pattern privacy.

In existing work, one proposed approach for ensuring client access pattern privacy (and confidentiality) tackles the case of a single-owner model. Specifically, a *service provider* hosts information for a *client*, yet does not find out which items are accessed or what is stored. In this setup the client has full control and ownership over the data: other parties are able to access the same data through this client only. One prominent instance of such mechanisms is Oblivious RAM (ORAM) [Goldreich and Ostrovsky 1996]. For simplicity, in the following we will use the term ORAM to refer to any such outsourced data technique. This paper considers a single-owner storage model. Though the online services cited above support sharing between users, they all have a single-owner component.

One of the main drawbacks of existing ORAM techniques is their overall time complexity. Specifically, in real-world setups ORAM [Goldreich and Ostrovsky 1996] yields execution times of hundreds to thousands of seconds per single data access. Recent work, including this work, provide various mechanisms to improve the efficiency by orders of magnitude.

This Contribution. In this paper we propose to build on the work in [Williams and Sion 2008] to introduce an efficient ORAM protocol with significantly reduced communication and computation complexities. Our protocol uses the ORAM-based [Goldreich and Ostrovsky 1996] pyramid-shaped database layout and reshuffling schedule employed in [Williams et al. 2011]. This yielded a protocol with a communication complexity of $O(\log^2 n)$ in the presence of $O(\sqrt{n \log n})$ client working memory, for a database sized n. Here, however, we deploy a new construction and more sophisticated reshuffling protocol, to reduce both the communication complexity (to $O((\log n)(\log^2 \log n))$) and the server storage overheads (to $O(n)$)—yielding a comparatively fast and *practical* oblivious data access protocol.

Efficiency. One of the main intuitions here is to store each pyramid level as an encrypted hash table and an encrypted Bloom filter (indexing elements in the hash table). The Bloom filter allows the client to privately and efficiently—without scanning of $O(\log n)$ fake block buckets for each stored block to hide the success of each level query as in previ-

[1]This is rarely a problem in practice since the memory locations are not typically disclosed at a granularity useful to any adversary. However, studies of cache access pattern vulnerabilities [Neve and Seifert 2007] show the gravity of such situations when they are—and what the amount of information that can be leaked through only subtle information concerning the access pattern.

ous ORAMs—identify the level where an item of interest is stored, which is then retrieved from the corresponding hash table. Less server-side storage is required ($O(n)$ instead of $O(n \log n)$), thus increasing throughput and reducing required server-side storage by an order of magnitude.

Privacy. The approach guarantees client access pattern privacy, since the same operations are performed at all pyramid levels, in the same sequence for any item of interest. The use of the encrypted Bloom filters allows the client to query an item directly at each level without revealing the success, instead of relying on a series of $O(\log n)$ fake blocks for each stored block to hide the success of each level query. Our contributions consist also of a new reshuffling algorithm that obliviously builds and maintains the encrypted Bloom filters and of a more efficient oblivious merge-and-scramble.

Correctness. Moreover, authenticated per-level integrity constructs provide clients with *correctness* assurances at little additional cost, specifically ensuring that illicit server behavior (e.g., alterations) does not go undetected.

Moreover, our ORAM protocol is well suited for deployment on constrained hardware such as SCPUs. We propose its deployment on existing secure hardware (such as IBM 4764 [IBM 2006]) to implement Private Information Retrieval (Figure 2), and show that the achievable throughputs are practical and much higher than existing work. These results contribute key insights towards making PIR assurances practical.

2. MODEL

Deployment. We consider the following concise yet representative interaction model. Sensitive data is placed by a client on a data server. Later, the client will access the outsourced data through an online query interface exposed by the server. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. Without sacrificing generality, we will assume that the data is composed of equal-sized blocks (e.g., disk blocks, or database rows).

Clients need to read and write these stored data *block*s with correctness assurances, while revealing no information to the (curious and possibly malicious) server. We describe the protocols from the perspective of the client, who will implement two privacy-enhanced primitives: read(*id*), and write(*id*, *newvalue*). The (un-trusted) server need not be aware of the protocol, but rather just provide traditional store/retrieve primitives.

Adversary. The adversarial setting considered through Section 4 assumes a storage provider that is *curious and possibly malicious*. Not only does it desire to illicitly gain information about the stored data, but it could also attempt to cause data alterations while remaining undetected. We prove that clients will detect any tampering performed by the server, before the tampering can affect the client's behavior or cause any data leaks. We do not consider timing attacks, noting that an implementation can be turned into a timing-attack free implementation without affecting the running time complexity. We also do not address direct denial of service behavior.

Cryptography. We require three cryptographic primitives with all the associated semantic security [Goldreich 2001] properties: (i) a collision-resistant keyed hash function which builds a distribution from its input indistinguishable from a uniform random distribution, (ii) an encryption function that generates unique ciphertexts over multiple encryptions of the same item, such that a polynomially bounded adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or unique items, and (iii) a pseudo random number generator whose output is indistinguishable from a uniform random distribution over the output space.

Notation. n is used throughout to denote the database size in blocks. Communication complexity is represented in terms of words: that is, for simplicity, we assume that a block identifier of $O(\log n)$ bits can be transmitted at $O(1)$ cost (this creates parity between the communication complexity and computational complexity representations). We use log to represent the natural log.

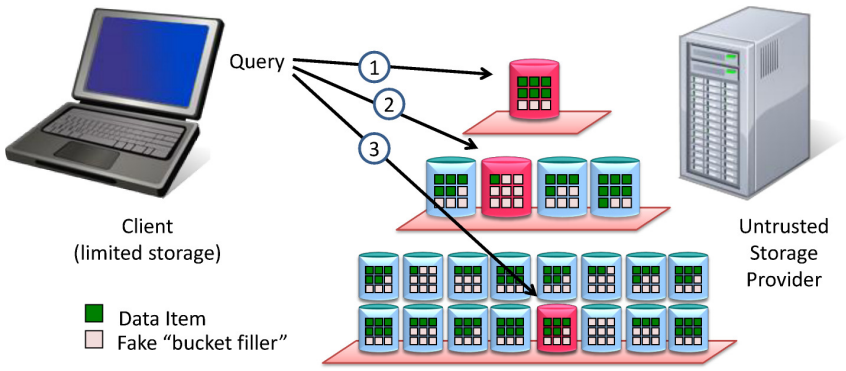

Fig. 1. Querying ORAM.

When we refer to a "hash function," we are referring to a cryptographic hash function as described above in (i). We refer the reader to Goldreich's book [Goldreich 2001] for an explanation of how a deterministic encryption function can be used to build a semantically secure encryption function.

Throughout the paper, we use a security parameter c, with the goal that the adversary's advantage at distinguishing access patterns be negligible in this security parameter. We do not distinguish the security parameter for the cryptographic functions (e.g., key lengths) from the security parameters for query privacy (e.g., Bloom filter sizes), since they are all related by constant factors.

3. RELATED WORK

3.1. Oblivious RAM

Oblivious RAM [Goldreich and Ostrovsky 1996] provides access pattern privacy to clients (such as software processes) accessing a remote database (or RAM), requiring only logarithmic storage at the client. The amortized communication and computational complexities are $O(log^3 n)$.

In ORAM, the database is a set of n encrypted blocks. Supported operations are read(id), and write(id, $newvalue$). The data is organized into $log_4(n)$ levels, as a pyramid. Level i consists of up to 4^i blocks; each block is assigned to one of the 4^i buckets at this level as determined by a cryptographic hash function. Due to hash collisions, each bucket may contain from 0 to $O(\log n)$ blocks.

ORAM Reads. To obtain the value of block id, the 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 (Level 0, 1 bucket) 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.* Figure 1 illustrates the ORAM query structure.

For (ii), once all levels have been queried, the client re-encrypts the query result with a different nonce 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 will be 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 (instead of the

old value). Inserts are simply writes of previously unassigned values. They are performed identically to writes, since no old value will be discovered in the query phase. Note that semantic security properties of the re-encryption function ensure the server is unable to distinguish between reads, writes, and inserts, since the access patterns are indistinguishable. This means the database size increases on every access (insert, write, or read).

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 different hash function (e.g., changing the "salt" value used with each input). Thus, accesses to this new generation of the second level will hence-forth be completely independent of any previous accesses. Each level overflows once the level above it has been emptied 4 times. Any re-ordering must be performed obliviously: once complete, the adversary must be unable to make any correlation between old block locations and 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.

ORAM Costs. Each query requires a total online cost of $O(\log^2(n))$ for scanning the $\log n$-sized bucket on each of the $\log n$ levels, plus an additional, amortized cost due to intermittent level overflows. Using a logarithmic amount of client memory, reshuffling levels in ORAM requires an amortized cost of $O(\log^3(n))$ per query.

In [Williams and Sion 2008] we introduced an ORAM-variant with a cost of $O(\log^2 n)$ assuming a small amount of client memory. An upper bound on this amount required was established in [Williams et al. 2011] at $O(\sqrt{n \log n})$. In these works, the assumed client memory is used to speed up the reshuffle process by taking advantage of the predictable nature of a merge sort on uniform random data. We build on these results here.

3.2. Private Information Retrieval

Another set of existing mechanisms handle access pattern privacy (but *not data confidentiality*) in the presence of *multiple mutually non-trusting clients*. Private Information Retrieval (PIR) [Chor et al. 1995] protocols aim to allow (arbitrary, multiple) clients to retrieve information from public or private databases, without revealing to the database servers which records are retrieved.

In initial results, Chor et al. [Chor et al. 1995] proved that in an information theoretic setting, any single-server solution requires $\Omega(n)$ bits of communication. PIR schemes with only sub-linear communication overheads, such as [Chor et al. 1995], require multiple non-communicating servers to hold replicated copies of the data. When the information theoretic guarantee is relaxed single-server solutions with better complexities exist; an excellent survey of PIR can be found online [Gasarch ; 2004].

Sion et al. showed [Sion and Carbunar 2007] that due to computation costs, use of existing non-trivial single-server PIR protocols on current hardware is still orders of magnitude more time-consuming than trivially transferring the entire database. Recent work [Olumofin and Goldberg 2011] has shown that *multi-server schemes* (in which servers are trusted not to collude) as well as more recent lattice-based mechanisms are now 10-1000 times faster than downloading the entire database. This still leaves PIR orders of magnitude behind ORAM mechanisms—which are faster at the expense of requiring client-side key management.

3.3. Secure Hardware-aided PIR

The recent advent of tamper-resistant, general-purpose trustworthy hardware such as the IBM 4764 Secure Co-Processor [IBM 2006] has opened the door to efficiently deploying

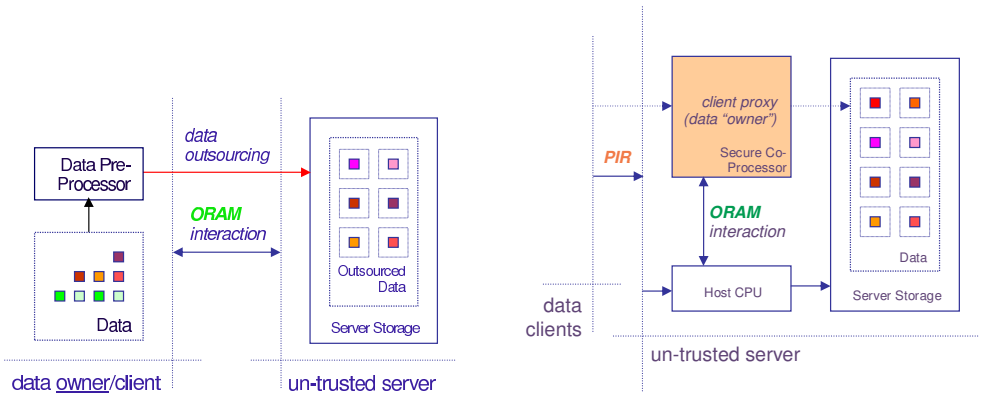

Fig. 2. (left) Simple ORAM Protocol between a client and a server. (right) A trusted server-side client proxy can be used to build a PIR interface on top of ORAM assurances.

ORAM privacy primitives for PIR purposes (i.e., for arbitrary public or private data, not necessarily originated by the current client) by deploying such hardware as a trusted server-side client proxy.

Asonov was the first to introduce [Asonov 2004] a PIR scheme that uses a secure CPU to provide (an apparent) $O(1)$ online communication cost between the client and server. However, this requires the secure CPU on the server side to scan portions of the database on every request, indicating a computational complexity cost of $O(n)$, where n is the size of the database.

An ORAM-based PIR mechanism is introduced by Iliev and Smith [Iliev and Smith 2004], who deploy secure hardware to achieve a cost of $O(\sqrt{n}\log n)$. This is better than the poly-logarithmic complexity granted by ORAM for the small database sizes they consider. This work is notable as one of the first implementations of ORAM-based PIR setups. Figure 2 summarizes the interaction between the client and server in ORAM, and how to turn an ORAM implementation into a PIR implementation using a Secure CPU.

An ORAM-based PIR mechanism with $O(n/k)$ cost is introduced in [Wang et al. 2006], where n is the database size and k is the amount of secure storage. The protocol is based on scrambling of a minimal set of server-hosted items. A partial reshuffle costing $O(n)$ is performed every time the secure storage fills up, which occurs once every k queries. While an improvement, this result is not always practical since the total database size n often remains much larger than the secure hardware size k. For storage of $k = c\sqrt{n\log n}$ (the amount of *working memory* assumed in this paper), this mechanism yields an $O(\sqrt{n/\log n})$ complexity (significantly greater than our $O((\log n)(\log^2 \log n))$ for practical values of n). Boneh et al. examine a similar construction in [Dan Boneh and Popa 2011], formalizing the security model.

3.4. Bloom filters

We include a brief definition of Bloom filters here since we will rely on them in our construction. Bloom filters [Bloom 1970] offer a compact representation of a set of data items. They allow for relatively fast set inclusion tests. Bloom filters are *one-way*, in that, the "contained" set items cannot be enumerated easily (unless they are drawn from a finite, small space). Succinctly, a Bloom filter can be viewed as a string of b bits, initially all set to 0. To *insert* a certain element x, the filter sets to 1 the bit values at index positions $H_1(x), H_2(x), \ldots, H_h(x)$, where $H_1, H_2, \ldots, H_h$ are a set of h hash functions. Testing set inclusion for a value y is done by checking that the bits for *all* bit positions $H_1(y), H_2(y), \ldots, H_h(y)$ are set.

By construction, Bloom filters feature a controllable rate of false positives r for set inclusion tests—this rate depends on the input data set size z, the size of the filter b and the number of cryptographic hash functions h deployed in its construction: $r = \left(1 - (1 - 1/b)^{hz}\right)^h$.

Encrypted Bloom Filters. An Encrypted Bloom filter, once we construct it obliviously, will allow us to quickly and obliviously check any level for the presence of the desired item. We shall make a set of four modifications to turn a Bloom filter into a remote data structure that we can query privately. First, we use keyed cryptograhic hash functions in place of public hash functions. Second, we store their bit representation encrypted while still allowing client-driven Bloom filter lookups. This is done by dividing the Bloom filter into constant-sized chunks, encrypting each chunk of bits with a semantically secure encryption algorithm, and attaching a message authentication code (MAC) for integrity. To prevent an adversary from returning the wrong chunk, the MAC must protect not just the encrypted bits, but also (e.g., as "additionally authenticated data") the location of this chunk (which includes the position of the chunk within the Bloom filter, and the level and generation of the Bloom filter itself). Third, we use an oblivious construction process that guarantees that the construction transcript appears independent of the contents and queries. Fourth, we sacrifice the constant time query cost in exchange for a guarantee that the possibility of a false positive is negligible.

3.5. Recent related work

There have been many new approaches to Oblivious RAM introduced in the past several years. We will review these new approaches, and show that the Bloom-filter-based approach described in this paper is as important now as ever. In particular, this ORAM construction finds new use as a fundamental building block in more recent ORAM constructions [Williams and Sion 2012a; 2012b].

A new approach to speed up ORAM was revealed by Pinkas et al. [Pinkas and Reinman 2010], showing the applicability of the cuckoo hash construction [Pagh and Rodler 2004]. Unfortunately, this was shown [Goodrich and Mitzenmacher 2011] to leak access privacy information. A similar, but secure, approach, allowing efficient item lookup while hiding success was developed [Goodrich and Mitzenmacher 2011]. The result is a $O(\log^2 n)$ solution requiring *logarithmic* client storage and requiring only $O(n)$ server storage. This approach has been re-applied in other work [Goodrich et al. 2011; Kushilevitz et al. 2011].

This research aims at the same goal as we do: obtaining efficiency in a level-based ORAM by decreasing the server storage and speeding up queries. We eliminate the expensive buckets on the server, reducing the storage cost (and as a result, the query cost and construction cost) by a factor of $O(\log n)$. The cuckoo hash construction mentioned above also eliminates the excessive server storage, resulting also in faster queries and level construction. While we use an Oblivious Merge Scramble that runs in $O(n \log \log n)$ time, and assumes $c\sqrt{n \log n}$ client memory, the mechanism of [Goodrich and Mitzenmacher 2011] uses a novel "Randomized Shell Sort" sorting network that runs in $O(n \log n)$ time and assumes only logarithmic client memory. Finally, unlike [Goodrich and Mitzenmacher 2011], we provide correctness guarantees against a fully malicious adversary. We do not claim that the cuckoo hash approach cannot be augmented to provide correctness, but rather that it does not follow trivially from the construction. In particular, construction of the cuckoo hash table requires accessing items a non-deterministic number of times, making it unclear how to detect rollback or data deletion attacks. We suggest that both techniques are relevant for different scenarios.

Researchers have recently recognized the utility of de-amortized constructions. [Kushilevitz et al. 2011], [Goodrich et al. 2011], and [Dan Boneh and Popa 2011], all provide de-amortized constructions. For simplicity, the construction described here is still amortized, but techniques similar to those described in [Goodrich et al. 2011] are applicable.

A promising recursive construction technique is introduced in [Stefanov et al. 2012] under the assumption of $O(n \log n)$ client storage. Using this storage, it promises to reduce the level construction cost with only $O(\log n)$ query cost. 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. This amount of storage makes it safe, e.g., to request an item from each level simultaneously, since the client already knows where the item is actually stored, resulting in a constant number of online round trips. 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 constant fraction of the outsourced storage. Their alternative construction presented, requiring only $O(\sqrt{n})$ storage, recursively uses a $\log_2 n$-round-trip ORAM to store this position map, at a query cost of $O(\log^2 n)$.

4. A SOLUTION

In previous work [Williams and Sion 2008] we achieved a complexity of $O(\log^2 n)$ in a protocol offering *access privacy* but *no correctness* assurances. Here we build on that result by deploying a new construction and more sophisticated reshuffling protocol, to significantly reduce both communication complexity to only $O((\log n)(\log^2 \log n))$ (amortized per-query), under the same assumption of $c\sqrt{n \log n}$ temporary client storage, while also endowing the protocol with correctness assurances.

4.1. Overview

Similar to ORAM (see Section 3.1 for more details), data is organized into $\log(n)$ levels, pyramid-like. Level i consists of up to 4^i items, stored on the server as label-value pairs. These pairs can be stored and retrieved in $O(1)$ time if the storage provider implements a suitable hash table such as the constant time, constant space hash defined in [Motwani and Raghavan 1995]. This differs from ORAM, which stores an item at level i using a keyed hash function to determine its storage bucket (of size $O(\log n)$, to allow for hash collisions) within the level. The use of *fixed-sized hash buckets in ORAM instead of a simple hash table adds a* $O(\log n)$ *storage overhead multiplier, and slows down query processing*, but the buckets are necessary; otherwise queries to a hash table could reveal whether the item was found at this level.

Here *we avoid the overhead of using buckets* to mask the query result by using Bloom filters [Bloom 1970] (constructed to be collision-free). Before attempting to query for an item that might not be at the current level, a per-level Bloom filter is queried first. The bits of the Bloom filter are encrypted, hiding the result of the query. If the Bloom filter indicates that the item is *not* at this level, we query the level for a unique fake item instead and continue with the next level. Once we eventually find the desired item (at a future level)—it will be moved into the top pyramid level—above the levels where it was searched for before (as in ORAM). This ensures that the same item will never be queried for in that instantiation of the Bloom filter again (as now it will be found higher in the pyramid, or a reshuffle would have been triggered).

Insight One: Faster Lookup. Thus one key insight in our mechanism is that we can construct an encrypted Bloom filter to perform set membership tests, without revealing the success of our query. Additionally, we design a novel construction procedure that assembles the encrypted Bloom filter without revealing any correlation between scanned items and associated Bloom filter positions. The final benefit of using encrypted Bloom filters is that all unique queries are computationally indistinguishable due to the nature of the keyed hash function used to index the filter. This allows us to modify ORAM with significant performance benefits, since we can avoid handling hash collisions, which add a $\log n$ factor in total database size as described above.

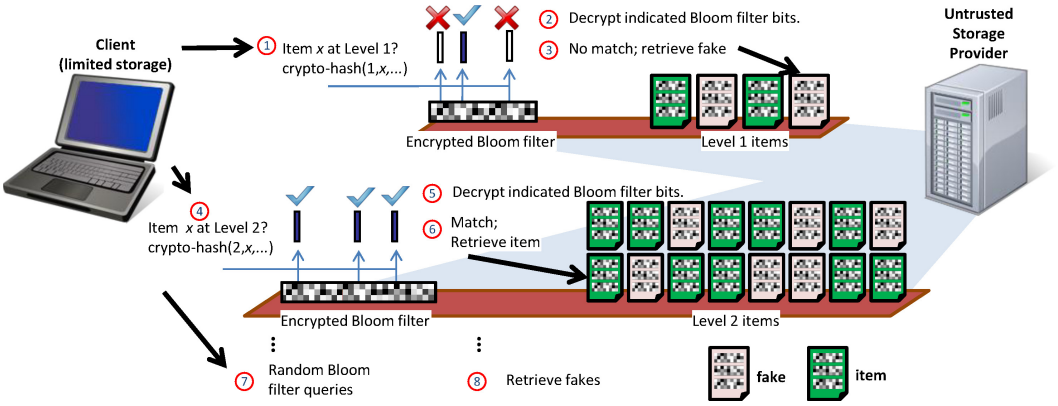

Fig. 3. Sample query. The client first checks the level 1 Bloom filter (1) to see whether the item is at Level 1. Finding at least one bit decrypting (2) to *0*, the client knows it is not. The client then asks for a fake item (3) at Level 1. This fake item was previously placed here for just such a scenario; the server does not know that it is fake. The client then queries the Level 2 Bloom filter (4). Seeing that all bits decrypt (5) to *1*, the client knows the item is stored at Level 2. It requests the item (6); again, the server does not know whether it is returning a fake or real item. On subsequent levels (7), the client now issues random Bloom filter queries and requests for fake items (8), to prevent the server from learning that the item was found at Level 2. Not illustrated in this figure is the next step, which places the discovered item back into the top level, and possibly initiates a reconstruction.

The notion of encrypting a Bloom filter has been studied previously, e.g. in [Bellovin and Cheswick 2004]. However, here we use a novel construction that hides the construction process, the inputs, and the results of the Bloom filter.

Insight Two: Correctness. Moreover, we deploy a set of authenticated, per-level integrity constructs to provide clients with *correctness* assurances at minimal additional cost. We specifically ensure that illicit server behavior (e.g., alterations) does not go undetected.

We now detail these components.

4.2. Query Processing

A query consists of a read or write request for a data item. These items are kept at the storage provider at a particular level; part of the client's job is to determine which level the item is at without revealing this to the server. Algorithm `Query` shows the pseudo-code of this operation; it is also illustrated in Figure 3.

To process a query, the client first downloads and scans the server-stored item cache (line 13) then proceeds to search each level, starting at the top (line 14). A labeling function consisting of a hash of the item ID with several level parameters ($fakeAccessCtr(level)$ and $Gen(level)$) generates the unique label by which the client can find the item at a particular level, if the item is indeed there. $fakeAccessCtr(level)$ represents the number of accesses to $level$ since the last reshuffle, and $Gen(level)$ represents the number of times $level$ has been reshuffled. The use of $fakeAccessCtr$ (line 14) ensures that successive queries request unique fake items, which the client knows are stored on the server. The use of $Gen(level)$ ensures that items on every subsequent reshuffle of a level have unique labels. Both functions are computed from the total number of accesses to the system thus far.

The search of any one level requires $O(\log \log n)$ time since the number of hash functions to request h is $O(\log \log n)$. If at the ith level the item has not already been found (in the cache or a previous level) (line 17) the client first computes the positions where the item would be stored in the ith level Bloom filter (line 18). It then retrieves those encrypted bits from the server-stored Bloom filter (line 20). If the decrypted bits are all *1* (line 22), the

client has found the item at the ith level. It then computes the label under which the item is stored in the level (referenced by its hash table) (line 23) and asks the server to remove and return the corresponding item (line 24). If at least one decrypted Bloom filter bit is *0* (line 26), the client instead performs the same operation using a *fake* label (built at line 14), known to be stored at the server (line 27).

Once the client has found the item (line 29), it proceeds by seeking fake items on the subsequent levels. This avoids revealing the level that answered the query, which would provide a correlation between queries. The client first searches a random set of positions (line 30) in the Bloom filter at the ith level (line 33), then asks the server to retrieve and remove a fake item from the level (line 34).

Note that the client queries the remotely-stored encrypted Bloom filter by requesting the encrypted values of positions indicated by the label function. While this reveals the requested Bloom filter positions to the remote server, *nothing is lost* as we prevent correlation by guaranteeing that any Bloom filter is never queried for any particular item more than once. Since the values at these positions are each encrypted, the server never learns the result of the Bloom filter query.

4.3. Access Privacy

The client achieves access pattern privacy by maintaining two conditions. First, *no item is ever queried twice using the same label*. This is achieved by removing the item, once it is found, and placing it in the item cache. Thus, on future queries the client will locate it in the item cache before repeating a label request; fake requests will be substituted on the lower levels. As items propagate out of the item cache (described in Section 4.4), the label functions are updated, so that the item has a different label by the time it makes it back down to a particular level.

Second, *the access patterns must appear indistinguishable* from random no matter where the item is located. A set of fake items is used to guarantee this: if the Bloom filter returns negative, indicating that the item is not stored at this level, a fake item from this level is retrieved instead.

On every single query, the server observes the same pattern. The client first scans the item cache, then queries a random value (chosen uniformly randomly, independently from all other information available to the server) from the level 1 encrypted Bloom filter—never queried by the client before. The server can observe the position of bits in the Bloom filter accessed, but it cannot observe whether each bit is set to *1* or *0*. The server then observes the client retrieve and delete one item from the level 1 hash table—never retrieved by the client before. This identical pattern of a random Bloom filter lookup followed by a random label-value retrieval and deletion continues through each level. Finally, the client appends a (semantically secure) encrypted value to the item cache.

Success or failure at each level is not revealed—the server cannot distinguish queries to fake entries in the Bloom filter from queries to real items in the filter, and the server cannot distinguish either of those from real items that are not in the filter. Additionally, the server cannot distinguish requests to real items from requests to fake items from the hash table, since the secure hash function used is non-invertible.

In addition to the constant amount of data transfer and computation exercised on each level by every query, we will see that $O(\log \log n)$ bits are examined to perform a Bloom filter lookup. Since there are $\log_4 n$ levels, the online cost per query is $O(\log n \log \log n)$ (measured in computation or transfer of words). Level reshuffling, described in the next section, will bring the total amortized cost per query of $O((\log n)(\log^2 \log n))$.

We now fill in the missing pieces: how to empty the item cache when it becomes full, and how to build the levels and the Bloom filters without revealing any information the server can correlate to retrievals.

```
1  server: Server                                              /* Server stub */
2  bits: int[ ]                          /* encrypted bit values of Bloom filter */
3  label, fakeLabel: int[ ]                                    /* search labels */
4  bfpositions: int[ ]                                         /* search labels */
5  fakeAccCtr: int[ ]                              /* per level access counter */
6  found: boolean
7  K: byte[ ]                                                  /* secret key */
8  h: int ← c log log n                  /* number of Bloom filter hash functions */
9  k: int                                 /* Bloom filter configuration parameter */
10 v: Object                                                 /* value for name x */
11 L: Object[ ] ← server.itemcache                          /* scan of item cache */
12 m_i: int                                                    /* Size of level i */
13 found, v ← L.search(x)                          /* Check recent queries first */
14 for i ← 1 to log_4 n do
15 |  fakeAccCtr[i]++
16 |  fakeLabel ← hash(i||'fake'||Gen(i)||fakeAccCtr[i]||K)
17 |  if found = false then
18 |  |  for j ← 1 to h do
19 |  |  |  bfpositions[j] ← hash(i||j||'BF'||Gen(i)||x||K)   mod k · h · m_i
20 |  |  end
21 |  |  bits ← server.getBloomFilter(i,bfpositions)
22 |  |  if decrypt(bits) = "11..1" then
23 |  |  |  label ← hash(i||'data'||Gen(i)||x||K)
24 |  |  |  v ← server.getAndRemove(label)
25 |  |  |  found ← true
26 |  |  else
27 |  |  |  server.getAndRemove(fakeLabel)
28 |  |  end
29 |  else
30 |  |  for j ← 1 to h do
31 |  |  |  bfpositions[j] ← random   mod k · h · m_i
32 |  |  end
33 |  |  server.getBloomFilter(i,bfpositions)
34 |  |  server.getAndRemove(fakeLabel)
35 |  end
36 end
37 itemCache.append(x,v)
38 return (v)
```

Procedure Query(x)

4.4. Handling Level Overflows: Reshuffle

The construction of the initial database structure is explained by the process of emptying the item cache: items are inserted into the item cache, which then overflows into the lower levels. Similar to ORAM, when the item cache is emptied, the contents are poured into level 1. In that process, level 1 and its new contents are reshuffled according to new label functions, removing any correlations between past and future lookups. When level 1 becomes full, it is poured into level 2, and so forth. Thus, the reshuffle process empties one level $i-1$ into the level i below it, which is four times as large as level $i-1$. Level i is then scrambled, hiding the correlation with the items' previous levels. A new Bloom filter for the lower level

is constructed—even items which happened to be at level i anytime in the past are now identified by a new unique label.

Let m be the size of the new level ($m \leq 4^i$). Let $h = O(\log \log m)$ be the number of hash functions used to generate a Bloom filter. The number of bits in the Bloom filter BF at level i will be equal to a constant k, times the number of hash function h and the number of items in the Bloom filter m. This ensures that any single bit in the Bloom filter is set with probability less than $1/k$. The database size n grows by one on each query, and thus varies over the lifetime of this level; h is chosen based on the value of $2n$ at the time of level construction, since n will never double during this period.[2] The constant k is a configuration parameter balancing a tradeoff between the number of Bloom filter hash functions and the Bloom filter size to meet the configured acceptable failure rate (e.g. 2^{-128}).

The client secret key is K. Let W be a working set stored on the server. Let L be a list of $O(m)$ entries, stored on the server. Let T be a $\sqrt{m}$ integer array stored at the client. Let Bkt be a server-hosted list of $O(\sqrt{m})$ buckets, of $\sqrt{m}$ entries each. Initially, W, L, T and Bkt are empty and all the bits in the server-stored BF are set to *0*.

In the next steps (steps 2 through 7 in the detailed description below, and illustrated in Figure 4) we build the encrypted Bloom filter without revealing the positions set in the Bloom filter.

A simpler two-step approach (with similar performance) is applicable if an oblivious *sort* is used in place of the Oblivious Merge Scramble. The idea is to replace steps 3-6 with an oblivious sort that puts the indices into the positions corresponding to segments; details are excluded here. However, any sort requires some extra computation (at least $O(m \log m)$ computation to sort m items instead of scrambling them). Such an oblivious sort can be obtained by replacing the client random array selection described in section 4.5 with a comparison that returns the smallest of the items in the fronts of the queue buffers. This does not affect the communication complexity, but it does slightly raise the mechanism's overall amortized *computation* complexity.

Constructing the Bloom filter in $O(m \log^2 \log m)$ time is accomplished by first scanning all items in the level, creating a list of what positions must be set in the Bloom filter to add each item, and storing this list encrypted on the server (step 2). To turn the list into a proper Bloom filter bit array, it will be sorted with a bucket sort—with $\sqrt{m}$ buckets of size $\sqrt{m}$, so that any bucket fits in private storage. To keep the buckets indistinguishable, we ensure they will all have the same size. Next (step 3) we calculate the size of each bucket, by scanning the list of Bloom filter positions, incrementing the appropriate bucket size tally for each position. In step 4 we add fake positions that will end up in those buckets that are lacking, according to the above (step 3) tally. Each bucket corresponds to a fixed range of positions in the final filter, so $j\sqrt{m}$ is used as a position that will wind up in bucket j. At this point the server will be able to identify the fake positions, since they are all at the end of the list. We then (step 5) scramble the list of positions to destroy all correlation between items and positions, and hide the fakes. In step 6 we move the scrambled positions into their buckets. Step 7 constructs the final Bloom filter, building a piece from each bucket. Steps 8 and 9 place the items in the new level while eliminating any correlation between the old and the new level structures.

The overflow process, performed by the client to pour level $i-1$ (or the item cache) into level i proceeds as follows (see Figure 4 for an illustration).

(1) **Merge levels.** Move all items from level $i-1$ and level i into W, a working buffer on the remote server. Discard the Bloom filters attached to both levels.
(2) **Build a list representation of the new Bloom filter.**

[2]Refer to Section 4.8 for discussion of why the database size increases by one on each query.

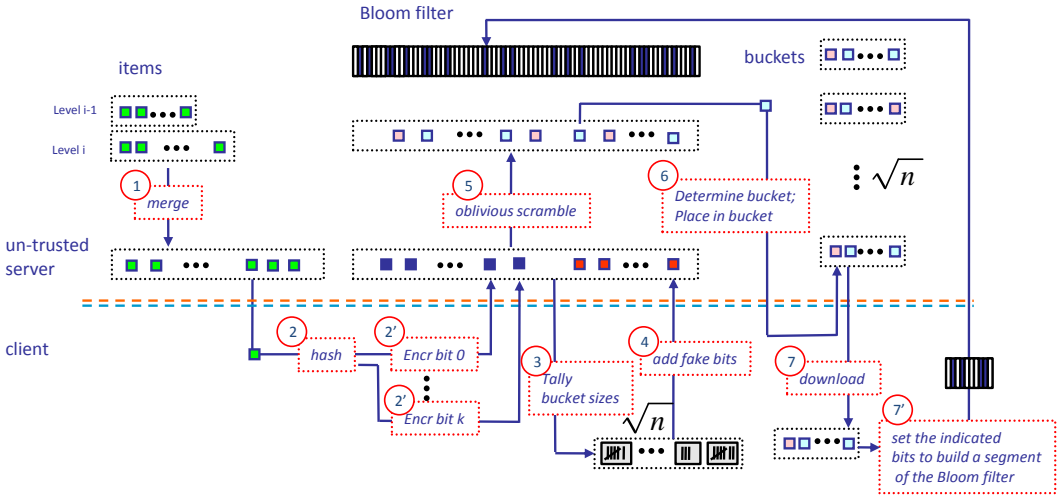

Fig. 4. Level reshuffle: Bloom filter construction (Steps 2 - 7)

Increment $Gen(i)$: the generation of level i has increased. Read each item $x \in W$ exactly once, and for each compute its $h = O(\log \log n)$ positions of bits $p_j(x) = hash(i||j||\text{'BF'}||Gen(i)||x.id||K) \mod k \cdot h \cdot m$, for $j = 1..h$, in the new Bloom filter. Encrypt each $p_j(x)$ separately and store all $E(p_j(x))$ values on the server-side list L. This step takes $O(m \log \log n)$ time, and $O(1)$ private (client-side) storage. *Costs here and in the following steps are expressed in terms of $m = 4^i$, the size of the current level.*

(3) **Tally Bloom filter positions to determine future bucket sizes.** Read each entry of L (the list of encrypted Bloom filter positions prepared in the previous step) exactly once. At the client side, for each entry $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, increment $T[idx(p)]$. This step allows the client to compute the number of bit-positions of L that will later (Step 6) end up in the Bkt structure. Effectively, the client builds a tally in local storage calculating the future size of each of the $\sqrt{m}$ buckets that will be built on the server in the step 6 bucket sort. We use $\sqrt{m}$ buckets of size $\sqrt{m}$ so that each bucket will fit in private storage in step 7, and the tally built here, with one counter per bucket, also fits in private storage. The step requires $O(m \log \log n)$ time and $O(\sqrt{m})$ private storage. (To avoid redundant scans, this step can be merged with the previous).

(4) **Add fake bits to make the bucket sizes equivalent.** The local tally from step 2 indicates the size of the largest bucket. We scan the tally, adding fake encrypted positions to the server-side list of encrypted positions as we go, so that all the buckets will have the same size as the largest bucket after the step 5 bucket sort. To add a fake position that will correspond to bucket j, the position $j\sqrt{m}$ is added to the list. (A simple balls and bins result predicts that the $\sqrt{m}$-sized buckets all have similar sizes, already; the number of fakes to add is small compared to the number of real items). Let max be the index of T such that $T[max] = max_{j=1}^{\sqrt{m}} T[j]$. For each $j = 1..\sqrt{m}$ generate $T[max] - T[j]$ fake values v_l such that the $\log m/2$ most significant bits of each v_l are equal to j. Store the encrypted, $E(v_l||\text{'fake'})$ value in L. This operation ensures that all the buckets of Bkt will store the same number of elements. The bucket size tally is discarded after this step. This step requires $O(m \log \log m)$ time.

(5) **Obliviously scramble the list of Bloom filter positions.** The encrypted indexes (the bit-positions of L, including the fakes) are scrambled, according to our Oblivious

Merge Scramble Algorithm, which destroys all correlation between the old positions and the resulting positions, which are a new uniform random permutation. The new list L stores the scrambled values. The algorithm requires $O(m \log^2 \log m)$ time and $O(\sqrt{m \log m})$ private storage.

(6) **Bucket-sort the list of Bloom filter positions.** For each $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, do $Bkt[idx(p)].add(E(p))$. Here the Bloom filter's scrambled, encrypted positions are bucket-sorted. The client retrieves each bit index, decrypts it to read it, and writes the encrypted value back to the bucket on the server corresponding to the $\frac{\log m}{2}$ most significant bits of the position. The bucket sort allows us to construct the encrypted Bloom filter in the next step without revealing to the server which bits are set: if we were to simply scan the entire list of positions setting the corresponding bits to true, the server would observe the bit flips in our encrypted array and learn what positions are set. The bucket sort groups related positions together so that we can build the Bloom filter from left to right in a single pass. This step requires $O(m)$ time.

(7) **Construct Bloom filter.** For each $j = 1..Bkt.size$, download $Bkt[j]$. Note that the size of $Bkt[j]$ is $\sqrt{m}$. Let $BF[j\sqrt{m}] \dots BF[(j+1)\sqrt{m}]$ be the segment of the Bloom filter corresponding to $Bkt[j]$, where $BF[idx]$ denotes the idxth bit of BF. For each $E(p) \in Bkt[j]$, let x be the least significant $\log m/2$ bits of p. Do $BF[j\sqrt{m} + x] = 1$. Store $E(BF[j\sqrt{m}]) \dots E(BF[(j+1)\sqrt{m}])$ on the server. Finally, store the oblivious Bloom filter of the working set W on the server.
Here the client downloads each bucket (which conveniently fits into local storage). The bucket corresponds to a $\sqrt{m}$-sized segment of the final Bloom filter—all positions in this bucket refer to a bit in this segment of the filter. The bits corresponding to listed positions are set to true, with all other bits set to false, in the local copy. The client encrypts this Bloom filter segment and uploads it to the server. Observe that the server has no indication of how many bits are true in this segment (other than that it is limited by the bucket size), nor which are true. The Bloom filter is finished at the end of this step. This step requires $O(m)$ time, and $O(\sqrt{m})$ private storage.

(8) **Scramble the items.** Finally, the client uses the Oblivious Merge Scramble Algorithm to scramble the actual items in the working buffer W. The Oblivious Merge Scramble requires $O(m \log \log m)$ time and $O(\sqrt{m \log m})$ private storage.

(9) **Add items back to level i.** Once scrambled, the items inserted under their new labels, according to the new labeling function for level i. For each item in $x \in W$ let $label(x) = hash(i||\text{'data'}||Gen(L_i)||x.id||K)$. Insert the pair $(x, label(x))$ into the set of items stored at level i. Add m fake items to level i, so that a query that turns out not to be for this level will have an item to retrieve instead (most of these m fakes will be deleted by the query process before the next reshuffle).

Level $i-1$ is now empty, and level i now contains all the items that were in level $i-1$. If level i is now full, this is then repeated as level i is then dumped into level $i+1$; the process is repeated recursively.

This procedure shows how level $i-1$ can be dumped into level i at a cost of $O(m \log^2 \log m) = O(4^i \log^2 i)$. Level $i-1$ is emptied once every 4^{i-1} queries, thus resulting in an amortized cost per query due to reshuffling of

$$\sum_{i=0}^{\log_4 n} O(\frac{4^i \log^2 i}{4^{i-1}}) = \sum_{i=0}^{\log_4 n} O(\log^2 i) = O((\log n)(\log^2 \log n))$$

As described below, client memory sized $c\sqrt{m \log m}$ is required for this procedure. The most stringent requirement is for the construction of the bottom level, when $m = n$. This case requires $c\sqrt{n \log n}$ memory.

The positions of the Bloom filter bits retrieved to check an item will appear to be chosen uniformly random, and completely independently of each other; therefore, any bit pattern indicates nothing about the query or the success of the query. They are independent of the bucket sort write pattern, which is the only other piece that could be tied to it, since the bucket sort write pattern is the only access pattern that varies during the reshuffle. The bucket writes are all identical except for the order of the writes, which is uniform random because of the scramble. The scramble has no bearing on the Bloom filter access pattern, which is dependent only on the query and the current Bloom hash function. Therefore the Bloom filter construction process yields no information about the items to the server in the resulting Bloom filter.

The level reorder process results in a new level that has no correlation to the old level, since the new permutation is chosen uniformly randomly (Theorem 4). The scramble process itself reveals no information about the new or old permutations, since the scramble has the same access pattern in all instantiations.

4.5. Oblivious Scramble Algorithm

To complete step 5 above, we now describe an algorithm that performs an oblivious scramble on a array of size m, with $c\sqrt{m \log m}$ local storage, in $O(m \log \log m)$ time with high probability. This is based on an algorithm by defined in [Williams and Sion 2008], which scrambles an array obliviously in time $O(m \log m)$ by ways of a merge sort. Since our application only requires a scramble, and not a complete sort, we can improve the asymptotic complexity by merging multiple arrays at once.

Informally, the algorithm is still a merge sort, except a random number generator is used in place of a comparison, and multiple arrays are merged simultaneously. The array is recursively divided into subarrays, which are then scrambled together in batches. The time complexity of the algorithm is better than merge sort since multiple subarrays are merged together simultaneously. Randomly selecting from the remaining arrays avoids comparisons among the leading items in each array, so it is not a comparison sort.

The Oblivious Scramble Algorithm proceeds recursively as follows, starting with the remote array split into subarrays of size $s = 1$, a security parameter c, and an array to scramble of size m.

(1) For subarrays sized s, allocate $\lceil \sqrt{m/s} \rceil$ buffers of size $c\sqrt{s \log m}$ (requiring $c\sqrt{m \log m}$ space total)
(2) Split the array into batches of $\sqrt{m/s}$ subarrays.
(3) For each of the $\frac{m/s}{\sqrt{m/s}} = \sqrt{m/s}$ batches:
   — Obliviously merge the subarrays in this batch together into one new subarray of size $(s)\sqrt{m/s} = \sqrt{ms}$, by performing the Oblivious Merge Step on the allocated buffers. The Oblivious Merge Step requires $c\sqrt{s \log m}$ local working memory for each of the $\sqrt{m/s}$ buffers, for a total of $c\sqrt{m \log m}$ working memory, and operates in $O(\sqrt{ms})$ time.
(4) In the end there are $\sqrt{m/s}$ subarrays of size $\sqrt{ms}$. Recurse with this larger subarray size.

One recursion of this algorithm requires a single pass across the level, costing $O(4^i)$ for level i. Each pass brings the total number of subarrays from m/s to $\sqrt{m/s}$, and we repeat until there is one subarray left. After iteration p, the number of subarrays remaining will be $m^{1/2^p}$. There will be 2 subarrays left when $p = \log_2 \log_2 m$. Since it takes $\log_2 \log_2 m$ passes to go from m to 2 subarrays, and each pass involves a single read and write of the entire array, the total running time / communication complexity for running the oblivious scramble on an array sized m is $O(m \log \log m)$.

We now describe the last remaining piece of the Oblivious Merge Scramble Algorithm, the Merge Step.

4.6. Oblivious Merge Step

The Oblivious Merge Step, whose pseudo-code is included here, takes r arrays of size s, and merges them randomly into a single array of size rs, preserving the ordering among the input arrays in the output arrays: if an item a is before item b in original array i, it will also be before b in the final array.

The permutation is chosen uniformly randomly out of all permutations that preserve the ordering of the original input items. To ensure this, we will take rs steps, choosing an item from the front of one of the r arrays at every step. The choice is biased since we choose each item *without replacement* randomly from the remaining items. If a particular array has a items left at step j, it has a $\frac{a}{rs-j}$ chance of being chosen at this step. In the pseudo-code we use the procedure `RandomlyChooseASubarray` for this purpose.

Implementation of such a random function is not quite trivial, so we also include its description. Given a set of partially filled arrays the `RandomlyChooseASubarray` procedure chooses, in $O(1)$ expected time, using $\sqrt{sr} \leq \sqrt{n}$ local storage, one of the arrays such that the probability of being chosen is proportional to the number of items in the array.

```
1  B ← New remote destination buffer size rs
2  t ← size of local queues, 2c√(s log m)
3  for i = 1 to r do
4  |  q_i ← empty queue stored locally, size t
5  |  for x = 1 to t/2 do
6  |  |  enqueue(q_i, decrypt(readNextItemFrom(A_i)))
7  |  end
8  end
   /* At this point, each queue will have t/2 items                    */
9  for x = t/2 to rs + t/2 do
10 |  if  x ≤ rs then
11 |  |  for i = 1 to r do
12 |  |  |  enqueue(q_i, decrypt(readNextItemFrom(A_i)));
13 |  |  end
14 |  end
   |  /* Now we've read r items; time to output r items                 */
15 |  for i = 1 to r do
16 |  |  v ← RandomlyChooseASubarray
17 |  |  x ← dequeue(q_v)
18 |  |  writeNextItemTo(B, encryptWithNewNonce(x))
19 |  end
20 end
21 return B
```

Procedure ObliviousMergeStep($A_1, \ldots A_r$)

RandomlyChooseASubarray Algorithm. This algorithm returns a random permutation of a sequence of length sr consisting of r values each repeated s times. All permutations are chosen with equal likelihood.

For first $sr/2$ selections, a random number is simply chosen from $1 \ldots n$. The number space is divided into r equal sized segments; the number will fall within the range corresponding

to array i. If it falls within the first part of array i, corresponding to one of the remaining items in this array, decrement this index for array i, and return i. Else, repick. This allows us to pick items without replacement, in expected $O(1)$ time, with only $O(r)$ working storage (to record the remaining items left in each array).

This will succeed in an average of 2 tries for the first $sr/2$ choices. After this, the target range must be decreased so that the average number of tries required does not increase. We simply change to pick random numbers from $1 \ldots sr/2$, again divided into r equal sized segments. This reduction can be repeated since the arrays are picked from with roughly equal frequency; the discrepancy between any two arrays will never pass $c\sqrt{s}$ as shown in the following theorem. Once the arrays are down near size $c\sqrt{s}$, they can no longer be divided into equal-sized arrays that are dense enough to successfully pick in a small number of tries. However, at this point they are now small enough to fit in local memory, so a Fisher-Yates scramble can be performed on the remaining items.

Analysis of the Oblivious Merge Scramble. The key to obliviousness is that we accomplish this random selection without affecting the actual access pattern of reading from the server. In [Williams and Sion 2008] this is implemented for 2 arrays; we now extend this to merge r arrays. By simply reading the input evenly at a fixed rate, and outputting the items indicated by the random function, the uniform nature of the random function will cause the output rates to be very similar with high probability.

In other words, we maintain a series of caching queues that are fed at a certain rate. According to a random function, we remove items from the queues. By the nature of the random selection, with high probability the queues will never overflow from being dequeued too slowly, nor empty out from being dequeued too quickly, as shown in Theorem 7.

Now, to show that the Oblivious Merge Scramble produces output indistinguishable from a uniformly randomly chosen permutation (from here on referred to as a "random permutation"), we start with a set of theorems proving that a single Oblivious Merge Step outputs a random permutation of the contents of its input queues. We break this into three steps. First we show that choosing a random permutation is equivalent to selecting according to a random interleaving of the input queues (Theorem 1). Second, we show that given random coin flips, the `RandomlyChooseASubarray` algorithm produces such a random ordering of the input queues (Theorem 2). Theorem 3 shows the combination of these procedures produces a random permutation of the elements of the input arrays.

Theorem 4 shows that these constructions can be combined to produce a uniform random permutation of a single array. Finally, we show that the ability to distinguish the output permutation from a random permutation reduces to the ability to distinguish these coin flips from random (Theorem 5).

THEOREM 1. *Selection from a set of randomly permuted input queues, according to a random interleaving of the input queues, produces a random permutation of all the elements of the input queues. In particular, given input queues $I_1 \ldots I_r$ of unique elements whose contents have been randomly permuted, removal from the queues according to a random permutation of the sequence $1^{|I_1|}2^{|I_2|} \ldots r^{|I_r|}$, (the "interleaving"), produces a random permutation of all the elements of $I_1 \ldots I_r$.*

PROOF. Since each output element belongs to a single unpermuted input queue, each output permutation is generated by a unique set of input queue permutations and interleaving. We identify the reverse mapping from output to input thusly: each input queue permutation is obtained by taking the ordering in the output array of the elements belonging to the given input array. The interleaving is obtained by taking the input queue identifier of each element in the output array. It follows from this one-to-one mapping of all output permutations back to the unique input that generates it, that all output permutations are possible.

Moreover, all possible input array permutations are independent and equally likely (by assumption). All possible interleavings are also equally likely (also by assumption).

Because there is a one-to-one mapping from inputs to the outputs, all outputs are possible, and all combinations of input permutations and interleavings are equally likely, it follows that all output permutations are equally likely. □

We now show that we can build a random interleaving of the r subarrays by repeatedly removing and outputting the head of one of the r subarrays selected randomly, with probability weighted according to the number of elements left in each subarray. More specifically, we show that the order of subarray selection, denoting subarray i by the integer i, is a random permutation of the sequence $1^s 2^s \dots r^s$, with all permutations equally likely.

THEOREM 2. *Choosing elements of a sequence S of rs integers 1 through s, with element S_i (for i from 0 through $rs-1$) chosen according to the probability density function $p(S_i = x) = (r - \sum_{j=0}^{i-1}(S_j = x?1 : 0))/(rs - i)$, results in a random permutation of the sequence $1^s 2^s \dots r^s$.*

PROOF. We show that all sequences are equally likely to be chosen this way. First, we show that swapping any two consecutive elements in any given sequence S results in an equally likely sequence.

Take consecutive positions i and $i+1$ in a sequence S, with elements from $S_i = k$ and $S_{i+1} = l$, where $k \neq l$. The chance $P(S)$ of generating S using the incremental construction is $p_{0,i-1}P(S_i = k|S_0..S_{i-1})P(S_{i+1} = l|S_0..S_i))p_{i+2,rs-1}$, where $p_{0,i-1}$ is the probability of choosing the portion of the sequence leading up to position i, according to the incremental definition. $P(S_i = k|S_0..S_{i-1})$ is the chance that a sequence starting as such will be followed by k, and is obtained from the definition, $(r - \sum_{j=0}^{i-1}(S_j = k?1 : 0))/(rs - i)$. Letting N_k be the number of times the element k appears in the subsequence $S_0..S_{i-1}$, this simplifies to $(r - N_k)/(rs - i)$. Likewise, $P(S_{i+1} = l|S_0..S_i)$ is seen to be $(r - N_l)/(rs - i)$, letting N_l similarly be the number of times the element l appears in the subsequence $S_0..S_{i-1}$. Note that N_l is also the number of times l appears in $S_0..S_i$, since $S_i \neq l$. We define $p_{i+2,rs-1}$ as the product of the probabilities of each choice in the rest of the sequence, $S_{i+2}..S_{rs-1}$. This allows us to express the likelihood of S as $P(S) = p_{0,i-1}\frac{r-N_k}{rs-i}\frac{r-N_l}{rs-i}p_{i+2,rs-1}$.

Consider now the sequence S' obtained by swapping positions i and $i+1$. Since $S'_0..S'_{i-1} = S_0..S_{i-1}$, the probability of obtaining $S'_0..S'_{i-1}$ is still $p_{0,i-1}$. The probability that prefix subsequence is to be followed by l is now $P(S'_i = l|S'_0..S'_{i-1}) = P(S'_i = l|S_0..S_{i-1}) = (r-N_l)(rs-i)$. The probability of choosing $S'_{i+1} = k$ is $P(S'_{i+1} = l|S'_0..S'_i) = (r-\sum_{j=0}^{i}(S'_j = l?1 : 0))/(rs - i) = (r - N_k)(rs - i)$. Finally, we see that the choice of the elements at the end of the sequence, $S'_{i+2}..S'_{rs-1}$ is unaffected by the order of elements at S'_i and S'_{i+1}. This is because the choice of each element depends only on the number of times each element precedes it, not the order of those elements. Thus, $P(S') = p_{0,i-1}\frac{r-N_l}{rs-i}\frac{r-N_k}{rs-i}p_{i+2,rs-1} = P(S)$.

Finally, observe that any permutation can be obtained by repeatedly swapping consecutive elements. Since we are equally likely to generate any two sequences that have a single pair of swapped consecutive elements, this implies an equal likelihood of generating any permutation. □

THEOREM 3. *Running the Oblivious Merge Step on unbiased random coin flips and randomly permuted input arrays generates a random permutation.*

PROOF. This follows from Theorems 1 and 2, and the definitions of the Oblivious Merge Step and `RandomlyChooseASubarray` algorithms: the Oblivious Merge Step selects a random permutation of input queues, according to an interleaving provided by

`RandomlyChooseASubarray`. Theorem 2 shows this interleaving to be random, and Theorem 1 shows this output to be a random permutation. □

THEOREM 4. *Given unbiased random coin flips, the Oblivious Merge Scramble produces a permutation selected with equal probability from all possible permutations of inputs.*

PROOF. The scramble consists of a series of array merge-scrambles, starting with arrays of size 1 on the first merge and resulting in an array of size n after the last merge.

We use an inductive argument on the merge step j.

As a base case, the inputs to the first merge step ($j = 1$) are arrays of size 1. These arrays are already in the only possible permutation, so by definition, this is a permutation selected uniformly randomly from all possible permutations.

Assume as the inductive hypothesis that the input to step j is a set of random permutations of the input arrays.

The Oblivious Merge Scramble produces as the output of step j (the input to step $j+1$) the set (referred to as a "batch of subarrays" in the Oblivious Merge Scramble definition) of outputs of Oblivious Merge Step, which is run multiple times across a partition of the output arrays from step j. By Theorem 3, each Oblivious Merge Step instance outputs a permutation of the elements of those input arrays selected uniformly randomly. Thus the input to step $j + 1$ is a set of randomly permuted arrays.

By induction, the output of any merge step past $j = 1$ is a set of randomly permuted arrays. In particular, the output the final merge step ($j = \log \log n$), which is a set containing a single element, consists of a randomly permuted array. □

Now we show that replacement of the source of randomness can be safely replaced with a PRNG source indistinguishable from random. That is, we will show that an adversary with an advantage ϵ at distinguishing the output array from a random permutation of the contents of the input arrays also has an advantage ϵ at distinguishing the PRNG output from random.

THEOREM 5. *(Replacement of the random source with a source* indistinguishable *from random.) The advantage of an adversary at distinguishing the output of Oblivious Merge Scramble from a uniformly randomly chosen permutation of the elements in the Merge Scramble input is equal to the advantage of the adversary at distinguishing the coin flips used in Merge Scramble from random.*

PROOF. An adversary B, given a source of coin flips, can distinguish it from uniform random as such: simulate the Merge step procedure using the set of coin flips; run A on the output. The advantage of adversary B at distinguishing the set of coin flips from uniform random is equal to the advantage of A at distinguishing the merge step output from random.

Take an algorithm A with advantage ϵ at distinguishing the Oblivious Scramble output from a true random permutation. This allows construction of an algorithm B that distinguishes the output of the employed PRNG from random, as follows. Algorithm B is given a sequence of bits S, with the goal of outputting 0 if it believes they are random, and 1 if it believes they are from the PRNG. Algorithm B takes the input and runs the Oblivious Merge using the bit sequence S, to produce a permuted array P. It then returns $A(P)$.

By definition, the probability that $A(P) = 1$ for a randomly permuted array P is at least ϵ less than the probability that $A(P) = 1$ for an array constructed by the Oblivious Merge Step using bits from our PRNG. Since running the Oblivious Merge Step on random coin flips generates a random permutation, as shown in Theorem 3, the probability that $A(P) = 1$ when S is a random set of bits is at least ϵ greater than when S is chosen by our PRNG.

The advantage of the algorithm at distinguishing the permutation from random thus reduces to its advantage at distinguishing our PRNG output from random. □

We now show correctness of the probabilistic Oblivious Merge Step. As described above, the Merge Step starts with a warm up phase (input but no output) in which each of r local queues are half-filled with elements from the corresponding input array. Next, there is a series of iterations, which each read 1 element from each of the r arrays, and output r elements, chosen randomly from the input arrays, according to the random permutation. Finally, in a "cool down" phase (output but no input), the remaining elements are emptied out of the queues, again according to the permutation. The warm up and cool down phases have no chance of overflowing or prematurely emptying the queues; we just need to show that the queues stay within their bounds in the main phase.

We start with a Theorem about random walks, proved in [Williams et al. 2011], which will be used to model our queue sizes vs. time.

THEOREM 6. *Let $p < 1/2$. Let W be a random variable distributed uniformly on the set of all walks $\mathcal{W}$ of length z, starting at 0, and containing zp $+1$-steps and $z(1-p)$ steps of $-p/(1-p)$. For such a walk w, let $f(w) = 1$ if and only if w exceeds $c\sqrt{pz\log z}$. Let $F_W := f(W)$. Then for each $z > 1$, $P_{F_W}(1) \leq 4ze^{-\frac{c^2 \log z (1-p)^2}{16}}$ which is negligible for $c \geq 4\sqrt{\log(4)}/\sqrt{\log z}$.*

THEOREM 7. *The Oblivious Merge Step succeeds, with high probability: for any s such that $1 \leq s \leq m/2$, the chance that the queue buffers sized $c\sqrt{s \log m}$ overflow or underrun is negligible w.r.t. the security parameter c.*

PROOF. Each instance of the Oblivious Merge Step reads from the set of $r = \sqrt{m/s}$ input arrays sized for some $1 \leq s \leq m/2$ in a random order. This order is chosen according to a random permutation of $1^s 2^s \ldots r^s$, as shown in Theorem 2. The walk is length $rs = \sqrt{ms}$. The size versus time of any array in the merge step is a walk chosen according to a random permutation of s $(+1)$-steps and $s(r-1)$ steps sized $(-(1-r)/r)$.

Setting $p = 1/r = 1/\sqrt{ms}$ and $z = rs = \sqrt{ms}$, Theorem 6 shows that the chance of such a walk exceeding a bound of $\pm c\sqrt{s \log \sqrt{ms}}$ is negligible in c. Since $s < m$, $c\sqrt{s \log \sqrt{ms}} < c\sqrt{s \log m}$, and the buffers sized $c\sqrt{s \log m}$ suffice. □

THEOREM 8. *Any advantage of the adversary (server) at deciding any function of the permutation output by Oblivious Merge Scramble on input of a publicly known size,* gained by learning *the sequence of reads and write tuples of the form ("read" or "write",location,value), translates into a violation of the semantic security of the encryption.*

PROOF. The sequence of locations read and written by the oblivious scramble defined in Algorithm ObliviousMergeStep is deterministic for a given input array size and *independent of the permutation being performed*. The encrypted blocks are all equivalently sized, and *value* is always a fresh output of a semantically secure encryption algorithm.

Any knowledge gained about the permutation from the encrypted values directly implies the server's ability to distinguish between or correlate different instances of encrypted equally sized data values. This violates the semantic security of the encryption algorithm. □

4.7. Bloom Filter and Query Privacy

Recent analysis of our construction in [Pinkas and Reinman 2010] reminds us that any Bloom filter false positives result in an access pattern leak. Further, it was suggested in [Kushilevitz et al. 2011] that eliminating the possibility of false positives could result in a performance penalty. To address these observations, we show here that the construction avoids false positives with high probability.

A Bloom filter containing z items has two parameters: y, the number of hash functions used (the number of bits set per item in the filter), and x, the number of bits in the Bloom filter, yielding the false positive rate $r = \left(1 - \left(1 - \frac{1}{x}\right)^{yz}\right)^y$.

The number of items z is determined by the level size, at 4^i for level i. The number of hashes used y is set to the security parameter c. Finally, we set the size of the Bloom filter x to scale proportionally with z and the number of hashes y, at $4^i ck$ for a constant k. We show now that the probability of a false positive r for a single Bloom filter lookup with these parameters is negligible in the security parameter.

Observe, as [Pinkas and Reinman 2010] points out, that we are not restricting the number of Bloom filter hashes to minimize the false positive rate. This allows flexibility in finding the performance optimal tradeoff between Bloom filter size (determining construction cost) and the number of hash functions (determining lookup cost).

LEMMA 4.1. *A Bloom filter containing $z = 4^i$ items, using $y = c$ hashes, and with $x = 4^i ck$ bits has a false positive rate negligible in the security parameter c for $k > 1$.*

PROOF. The portion of bits in the Bloom filter that are set is upper bounded by $\frac{zy}{x}$, yielding the looser bound $r \leq \left(\frac{zy}{x}\right)^y$.

$$r = \left(1 - \left(1 - \frac{1}{x}\right)^{yz}\right)^y \leq \left(\frac{zy}{x}\right)^y = \left(\frac{4^i c}{4^i ck}\right)^c = k^{-c}$$

This quantity is negligible in c. □

With $x \geq cz$, the cost to obliviously construct a Bloom filter sized x is shown earlier to be $O(x \log\log x)$ (with sufficient storage); since x is proportional to z, the asymptotic performance is not affected by the Bloom filter. Moreover, the time required to query this Bloom filter is $O(c)$, which is constant for any chosen security parameter c.

However, we require $\log_4 n$ lookups (in different Bloom filters) to perform an access. $k^{-c} \log_4 n$ is not strictly negligible for a variable n. Thus, we instead require the failure probability per Bloom filter lookup to be within $k^{-c}/\log_4 n$, so that the union bound gives us a total probability of failure per *access* of k^{-c}.

To achieve this we revise the Bloom filter parameters. The number of items z is still determined by the level size, at 4^i for level i. The number of hashes used y is now set to $c + \log_k \log_4 n$. The size of the Bloom filter x should still scale proportionally with z and the number of hashes y, at $x = kyz = 4^i k(c + \log_k \log_4 n)$ for a constant k. The probability of a Bloom filter false positive with these parameters is negligible in the security parameter, now over a period of $\log_4 n$ lookups.

Since the value of n increases by one each query, n is not constant over the lifetime of a level instance. However, once $n > 2$, the database size will never again double during this period. Thus, choosing a number of hash functions h based on a value of $c + \log_k \log_4 2n < c + 1 + \log_k \log_4 n$ suffices to limit the probability of a false positive over the entire duration of a level, even as n is increasing.

THEOREM 9. *A set of $\log_4 n$ queries over a Bloom filter containing $z = 4^i$ items, using $y = c + \log_k \log n$ hashes, and with $x = 4^i k(c + \log_k \log_4 n)$ bits has a false positive rate negligible in the security parameter c for $k > 1$ over a period of $\log_4 n$ lookups.*

PROOF.

$$r \leq \left(\frac{zy}{x}\right)^y = \left(\frac{4^i(c + \log_k \log_4 n)}{4^i k(c + \log_k \log_4 n)}\right)^{c + \log_k \log_4 n} = k^{-c - \log_k \log_4 n} = \frac{k^{-c}}{\log_4 n}$$

Taking the union bound over $\log_4 n$ Bloom filter lookups, the probability of failure is bounded by $r \log_4 n \leq k^{-c}$. This quantity is negligible in c. □

The time to query this Bloom filter is $O(c + \log\log n)$. The time/communication required to construct this Bloom filter obliviously is $O(x \log\log x) = O(4^i k(c +$

$\log_k \log n) \log\log(4^i k(c+\log_k \log n)))$. Assuming a constant k, and amortizing over the shuffle period of 4^i queries, this simplifies to a shuffle cost per query of $O((c+\log\log n)(\log\log n+ \log\log((c+\log\log n))))$. For a constant security parameter c, and since $\log\log\log\log n < \log\log n$ for $n > 1$, this shuffle cost is $O(\log^2 \log n)$ (per level). The construction cost across all levels is $O((\log n)(\log^2 \log n))$. In summary, guaranteeing a failure rate negligible in the security parameter c across all $\log n$ lookups for a given query raises the amortized shuffle cost per query from $O(\log n \log\log n)$ to $O(\log n \log^2 \log n)$. Further, it raises the online lookup cost from $O(\log n)$ to $O(\log n \log\log n)$.

THEOREM 10. *The server learns nothing about the client access pattern by observing queries.*

PROOF. Theorem 4 shows that all items at a particular level are scrambled uniformly randomly. The Bloom filter construction algorithm also scrambles the positions uniformly randomly, so that no correlation can be maintained before and after the scramble. Semantic security of the encryption function ensures that the encoding of an item reveals nothing about its position after the scramble.

The use of keyed cryptographic hash functions guarantees that unique queries will appear uniform random; the server gains no additional information by watching the queries. The reshuffle schedule ensures that the same lookup is never run against a particular Bloom filter or level instance twice between scrambles; after an item is queried it is placed in the top level, where it will be detected the next time it is queried. As the levels overflow, reshuffling ensures that the item will have a new label when it reaches a level at which it had previously been located. Since the scrambles destroy all correlation, all queries therefore appear independent and uniform random to an observer.

The overall access pattern appears constant and predictable to the server; reshuffles happen at fixed, predetermined positions, and the only difference is in the uniformly random selection without replacement of the items queried at each level. □

4.8. Increasing Database Size

The database size increases by one on each query. This is because when a queried item is moved to the top, a fake item is effectively left in its place in the lower level. This is necessary to avoid revealing which level the item is being moved from. This property has the benefit that reads are indistinguishable from inserts.

Another way of reasoning about why the database grows on every query is to see that the level reshuffle process inserts a fake item to retrieve at each level for every query (over the projected lifetime of the level). The Query procedure removes one item from each level. These are the pre-established fake items, except at *the level where the item itself is taken.* So these inserted fakes will all be deleted—except for those that happen to be located at the level its corresponding query is answered at.

If this behavior is undesirable, it is simple to consolidate the database periodically without affecting the amortized complexity. For example, one option is to periodically read the entire database into a temporary server-stored array, Obliviously scramble the array, then add each item back to a fresh database. If this process is run every time the database size doubles, then the cost of eliminating built-up fake items is still less than the amortized query cost, while the total number of built-up fake items remains less than the number of real items. Any such process ends up revealing, of course, the number of fake items removed.

5. CORRECTNESS AND INTEGRITY

In this section we introduce a set of integrity constructs that endow the above solution with correctness assurances. Specifically, we would like to guarantee that any storage provider tampering behavior is detected. All of these constructs can be implemented efficiently, with

few or almost no overheads: (i) Message Authentication Codes (MACs) are added for all the stored items and Bloom filters; (ii) unique version labels for each item in the data covered by the MAC are added to prevent any replay-type attack in which the server incorrectly replies with a previously MAC-signed message; (iii) incremental, collision-resistant commutative checksums are added to checksum the item sets contained in each level, to prevent the server from hiding or duplicating items during the level reshuffle process.

For (i) we require a MAC function, such that the computationally bounded adversary has no non-negligible ability to construct any message, MAC pair $(M, MAC(M))$ for an M that did not originate at the client. Every item uploaded to the server is protected by such a MAC. (ii) is straightforward since level construction already labels items uniquely, and the non-repeating generation of each level is known by the client. For (iii), besides the requirement of being collision-resistant, it should be easy for clients to maintain and update a checksum of a set. In particular, when adding or deleting items, it should be possible to to so incrementally, without recomputing the entire checksum value.

The *incremental hashing* paradigm of Bellare and Micciancio [Bellare and Micciancio 1997] can be used to construct exactly such a checksum. Fix any cryptographic hash function h (viewed as random oracle) and a large prime p. To hash a set $B = \{b_1, \ldots, b_l\}$, we compute the product

$$H(B) := \prod_{i=1}^{l} h(b_i) \bmod p. \tag{1}$$

Note that this hash construction allows both for easy addition of item b (multiplication with $h(b)$) and removal of any b_i (multiplication by $(h(d_i))^{-1}$) without needing to recompute the hashes of all values $b_1, \ldots, b_l$.

It can be shown in the random oracle model [Goldreich 2001] that it is computationally infeasible to find two sets that have the same checksum; the hash function (1) thus forms an easy and efficient way to authenticate the set of items in a level:

THEOREM 11. *If the discrete logarithm problem in $\mathbb{Z}_p^*$ is hard it is computationally infeasible to find two sets $A \neq B$ with $H(A) = H(B)$.*

THEOREM 12. *A client interface correctly implementing the above integrity constructs will detect all incorrect server responses before they reveal any part of the access pattern, or return an incorrect answer to the user of the interface.*

PROOF. The proof is straightforward by construction. We enumerate all ways in which the server can violate correctness.

If the server does not respond to a query at all, the client will detect the denial of service, with the caveat that over any asynchronous communication line it may be impossible to determine who is responsible the denial of service, or whether the response may eventually arrive.

Construct (i) guarantees that if the server replies with to a query with data that is not accompanying a valid MAC for that data, the client will immediately detect the invalid MAC and return $\perp$, thus limiting the damage done to denial-of-service.

Thus, it now suffices to consider only those attacks in which the server responds with data authenticated by a valid MAC. If the MAC function is secure, this reduces to considering only attacks in which the server responds with out-of-date data that was originally sent from the client (e.g., a replay attack).

Since every correct server response in this protocol consists of one item initially uploaded by the client, the only type of tampering attack that remains is one in which the server returns the wrong item or version of items in any response.

Construct (ii) attaches a unique version ID to every subsequent version of an item. Moreover, at every step the client knows the exact unique version ID of the desired item a priori. The version id label is simply a combination of the desired item's label/id, the level id, and the reshuffle count of the level. Thus, any attack in which the server returns the wrong item or version of an item, the returned version ID (which is MAC-protected), will mismatch the expected version ID.

Finally, due to its incremental, collision-resistant nature, construct (iii) allows clients to properly authenticate the accumulated per-level writes/reads of items during the reshuffle phase. Specifically, at all points during the online query phase, the client maintains the incremental commutative checksum for a level—when a client removes/adds an item it also updates the checksum accordingly. Then, during the reshuffle phase, the level is scanned and copied at once. The client can now verify the previous operations by re-computing the checksum for the level and comparing it with the locally maintained one. □

6. PERFORMANCE ANALYSIS

We now discuss main performance projections based on the network traffic, computation, and disk accesses. considering the following configuration: disk throughput = 80 MBytes/sec; disk sector size = 4096 bytes; disks are rotational-latency-free solid state disks (SSDs); client symmetric encryption crypto throughput = 100 MBytes/sec (from benchmarks provided by [Lipmaa 2006]); network throughput = 1 GBit/sec; network latency = 5ms. We set the number of Bloom filter hash functions to 50, and determine from that the corresponding Bloom filter size required to establish a false positive rate of 2^{-128}. The construction cost amortized per query, plus the online cost per-query cost, is graphed against a growing database size in Figure 5.

This plot assumes an optimal implementation in which resource costs of level construction can be paid in parallel: the server disk writes simultaneously with the client encrypting data and sending it on the network. That is, instead of paying these costs sequentially, we assume that the there is a bottleneck formed by the most limiting resource (in this case, the server disk throughput). Aside from start-up costs, this is roughly accurate for an optimized implementation. Reads during level construction, for example, are all deterministic; the server can read ahead to keep the disk busy even while the client is processing other data.

We plot three costs and their sum. The online query cost reflects the cost of performing $\log_4 n$ Bloom filter lookups on the server and reading the resulting items. This includes reading of $50 \log_4 n$ disk sectors: one for each Bloom filter position indicated by the secure Bloom filter lookup hash function at each level. It also models the disk read, network transfer, and decryption of the $\log_4 n$ items themselves.

The Bloom filter construction cost models the *simplified* Bloom filter construction procedure, in which a merge Sort is used in place of the merge Scramble. The merge scramble has an equivalent I/O cost but requires slightly more client computation (negligible in this scenario). The plot models the bottleneck resource usage, in this case disk throughput, limiting at 80 MBytes/sec. As justified below, we assume the full disk throughput can be obtained, despite the use of scattered reads smaller than a sector in this construction.

The item Oblivious Merge Scramble cost is analogous: it represents the bottleneck cost of $\log_2 \log_2 n$ passes over the data at each level.

6.1. Discussion

The graph suggests an optimized implementation can achieve over 8 queries per second on a 1 terabyte database, and over 5 queries per second on a 10 petabyte database. This assumes, of course, the previously established $c\sqrt{n \log n}$ client storage.

We assume rotational-latency-free solid state disks (SSDs) because our sort requires scattered data reading. However, use of SSDs does not immediately bring the observed disk throughput up to the optimal disk throughput, for the following reason. Although the

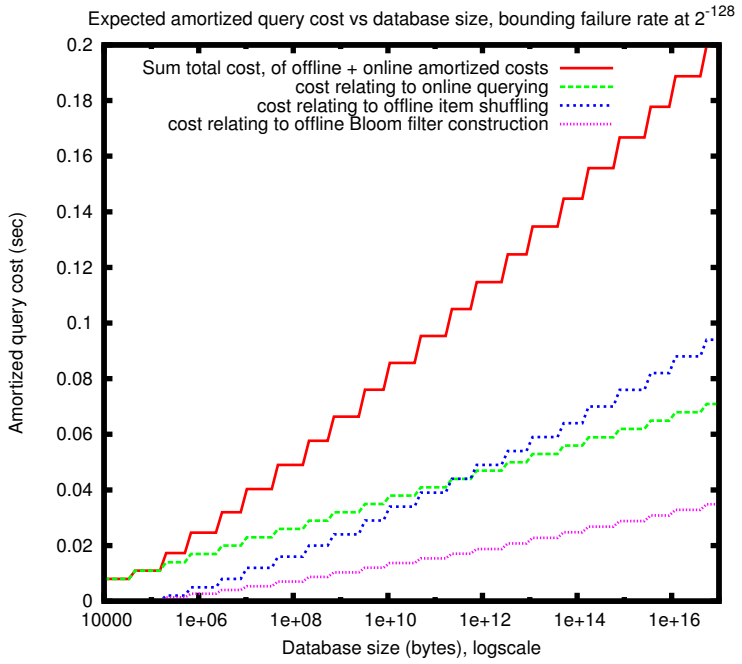

Fig. 5. Plot of the expected amortized query cost, broken down by source. The sort cost plotted is the bottleneck resource usage, summed over each of the $\log_4(x/\text{blocksize})$ levels.

disk throughput is roughly on par with the other potential bottlenecks of client encryption throughput and network throughput, the I/O operations speed of the disk comes into play here. That is, even though we assume a latency-free SSD, scattered reads and writes *smaller than a sector size* still have the cost of reading and writing the entire sector.

As an example, a somewhat tricky case to analyze arises once the data associated with Bloom filter construction is too big to fit in server memory. The Bloom filters themselves contain approximately 250 bits (≈ 50 of them set to *1*) for every item stored at the level. In our configuration, assuming Bloom filter segments of 16 bytes (each with a 16 byte MAC), this works out to only around 6.25 GB of Bloom filter data for a 1 TB database. However, the temporary lists used to scramble and build this same Bloom filter with privacy and correctness require a whopping 320 GB. Recall that ≈ 50 Bloom filter positions per level item are appended to a list, along with a MAC for each, and padded to be the same size as a 32-byte Bloom filter segment.

The apparent effect is that sorting this 320 GB list might require a full disk sector read and write per sort operation, when only 32 bytes are needed. This involves no additional penalty when sorting items as big as (or a multiple of) the disk sector size, but potentially cuts disk throughput by a factor of 128 when sorting our 32-byte Bloom filter positions. This penalty is imposed in most of the passes, for example, of a Randomized Shell Sort as in [Goodrich 2010]. On the other hand, the Oblivious Merge Sort described in [Williams et al. 2011] would not impose this penalty (or even an analogous disk seek penalty), since the data is read sequentially.

Where, then, does the Oblivious Merge Scramble stand? We show now that with intelligent caching and room for $\sqrt{n}$ sectors in server RAM, (e.g., 1 GB of RAM for a 1 TB database is more than enough) we can avoid this disk I/O penalty. In the first step of the Oblivious Merge Scramble, there are $\sqrt{n}$ items being merged together at once. In this case,

the entire contents of any single merge operation fit in RAM simultaneously, requiring only a single sequential read at the beginning and a single sequential write at the end.

Observe that the remaining merge steps (2 through $\log_2 \log_2 n$), have two properties. First, the subarrays are all size at least $\sqrt{n}$, which is greater than a disk sector for all potentially problematic database sizes. That is, databases smaller than, e.g., 4096^2 blocks, have Bloom filters small enough to be constructed entirely in server RAM. Second, the total number of subarrays is at most $\sqrt{n}$. Together, these properties mean that an entire sector (e.g. 4096 bytes) of every subarray can fit in server RAM. With careful disk cache use, and calibrating subarray sizes to a multiple of the disk segment size, we can read data sector-by-sector without wasting disk throughput. This means we can operate at the full SSD disk throughput. Note that this does not imply sequential reading of the data (and avoidance of the disk seek cost on a rotational disk).

7. CONCLUSIONS

In this paper we introduce a practical oblivious data access protocol with correctness. The key insights lie in new constructions and sophisticated reshuffling protocols that yield practical computational complexity (to $O((\log n)(\log^2 \log n))$) and storage overheads (to $O(n)$). This mechanism simultaneously provides *full computational privacy* and *correctness*.

While the construction described here can no longer be considered the most practical construction to date, it has proved to be an important advance enabling more modern ORAM constructions. To point toward the future directions of this work, we refer the reader to two forthcoming constructions that use the ORAM techniques introduced here as their fundamental building blocks. These ground-breaking constructions present new theoretic results (the first single-round-trip log-storage ORAM) and new practical results (the first Oblivious file system, and ORAM with unprecedented performance), respectively.

> We present SR-ORAM, the first single-round-trip polylogarithmic time Oblivious RAM that requires only logarithmic client storage. Taking only a single round trip to perform a query, SR-ORAM has an online communication/computation cost of $O(\log n \log \log n)$, and an offline, overall amortized per-query communication cost of $O(\log^2 n \log \log n)$, requiring under 2 round trips. The client folds an entire interactive sequence of Oblivious RAM requests into a single query object that the server can unlock incrementally, to satisfy a query without learning its result. This results in an Oblivious RAM secure against an *actively malicious* adversary, with unprecedented speeds in accessing large data sets over high-latency links. We show this to be the most efficient storage-free-client Oblivious RAM to date for today's Internet-scale network latencies.
>
> [Williams and Sion 2012b]

> Oblivious RAM throughput is increased by an order of magnitude, by enabling client parallelism in high-latency environments. This critical piece is missing from existing Oblivious RAMs, which typically require many communication rounds. A construction is next presented to enforce de-amortization across a class of Oblivious RAMs, drastically reducing the worst case query cost. These techniques are shown to be fundamental to the construction of any efficient Oblivious RAM. A high performance, fully functional implementation is then developed and analyzed, performing a query per second on a terabyte database over a 50ms latency link. Finally, the implementation is used to construct *privatefs*, the first Oblivious File System.
>
> [Williams and Sion 2012a]

REFERENCES

Amazon Web Services LLC. 2012. Amazon Simple Storage Service FAQs. Online at `http://aws.amazon.com/s3/faqs/`.

Asonov, D. 2004. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer Verlag.

Bellare, M. and Micciancio, D. 1997. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of EuroCrypt*.

Bellovin, S. M. and Cheswick, W. R. 2004. Privacy-enhanced searches using encrypted bloom filters. Tech. rep., Columbia University.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13,* 7, 422–426.

Canonical. 2012. Ubuntu One. Online at `http://www.canonical.com/consumer-services/ubuntu-one/`.

CBS. 2006. Feds Seek Google Records On Porn. Online at `http://www.cbsnews.com/2100-205_162-1221309.html`.

Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. 1995. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*. 41–50.

Dan Boneh, D. M. and Popa, R. A. 2011. Remote oblivious storage: Making oblivious RAM practical. Tech. rep., MIT. MIT-CSAIL-TR-2011-018 March 30, 2011.

Gasarch, W. A WebPage on Private Information Retrieval. Online at `http://www.cs.umd.edu/~gasarch/pir/pir.html`.

Gasarch, W. 2004. A survey on private information retrieval.

Goldreich, O. 2001. *Foundations of Cryptography*. Cambridge University Press.

Goldreich, O. and Ostrovsky, R. May 1996. Software protection and simulation on oblivious RAM. *Journal of the ACM 45*, 431–473.

Goodrich, M. and Mitzenmacher, M. 2011. Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In *38th International Colloquium on Automata, Languages and Programming (ICALP)*.

Goodrich, M., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. 2011. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Cloud Computing Security Workshop at CCS*.

Goodrich, M. T. 2010. Randomized shellsort: A simple oblivious sorting algorithm. In *Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

Google. 2012. GMail. Online at `http://gmail.google.com/`.

IBM. 2006. IBM 4764 PCI-X Cryptographic Coprocessor (PCIXCC). Online at `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`.

Iliev, A. and Smith, S. 2004. Private information storage with logarithmic-space secure hardware. In *Proceedings of i-NetSec 04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*. 201–216.

Kushilevitz, E., Lu, S., and Ostrovsky, R. 2011. On the (in)security of hash-based oblivious RAM and a new balancing scheme. Cryptology ePrint Archive, Report 2011/327. `http://eprint.iacr.org/`.

Lipmaa, H. 2006. AES ciphers: speed. Online at `http://www.cs.ut.ee/~lipmaa/research/aes/rijndael.html`.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms*. Cambridge University Press.

Neve, M. and Seifert, J.-P. 2007. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, E. Biham and A. Youssef, Eds. Lecture Notes in Computer Science Series, vol. 4356. Springer Berlin / Heidelberg, 147–162.

Olumofin, F. and Goldberg, I. 2011. Revisiting the computational practicality of private information retrieval. In *In Financial Cryptography and Data Security 11*.

Pagh, R. and Rodler, F. F. 2004. Cuckoo hashing. *J. Algorithms 51*, 122–144.

Pinkas, B. and Reinman, T. 2010. Oblivious RAM revisited. In *CRYPTO*. 502–519.

Sion, R. and Carbunar, B. 2007. On the Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*. Stony Brook Network Security and Applied Cryptography Lab Tech Report 2006-06.

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

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

Williams, P. and Sion, R. 2008. Usable Private Information Retrieval. In *Proceedings of the 2008 Network and Distributed System Security (NDSS) Symposium*.

Williams, P. and Sion, R. 2012a. A Parallel Oblivious File System. In *To Appear in ACM Conference on Computer and Communications Security (CCS)*.

Williams, P. and Sion, R. 2012b. Single Round Access Privacy on Outsourced Storage. In *To Appear in ACM Conference on Computer and Communications Security (CCS)*.

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

Williams, P., Sion, R., and Sotakova, M. 2011. Practical oblivious outsourced storage. *ACM Transactions on Information and System Security (TISSEC)*.

Xdrive. 2012. Online at `http://www.xdrive.com/`.