Asian Journal of Information Technology 15 (4): 781-790, 2016
ISSN: 1682-3915

Preserving the Privacy of Cloud Store Using Oblivious RAM

G. Sujitha, R. Tamil Mozhi and S. Sai Krishna
Rajalakshmi Engineering College, Anna University, 600025 Chennai, Tamil Nadu, India

Abstract: Security and privacy are major concerns that impact the shifting of data to a cloud. A widely-prevailing solution is to encrypt data before, it is moved to cloud storage. Nevertheless, encryption of data alone stored to be protected. An Oblivious RAM (O-RAM) comes to the rescue by concealing data access patterns in the cloud. This study describes the concept of O-RAM and proposes optimizations to reduce the volume of data transferred during data access, achieved by constructing a tree for storing contents in the cloud. During each access, the contents of buckets are evicted and re-shuffled. The results of the study demonstrate that greater the size of the tree, the lesser the number of blocks evicted from it.

Key words: Cloud privacy, security, oblivious RAM, binary tree, India

INTRODUCTION

Cloud computing has changed the very notion of computing as a service-on-demand. It has gained momentum, since it is a pay-as-you go model, offering the advantage of cost reduction through the sharing of computing resources as well as storage resources. The cloud computing network enables a user to access and manipulate information stored on remote servers. Storage as a service allows a client to outsource data to the cloud. Client-stored data in the cloud ought at the very least to be encrypted (Chor *et al*., 1998). Unfortunately, however mere encryption alone is insufficient. Encryption is not always enough to ensure privacy for outsourced data for, even if the data is in encrypted form the cloud provider can gain access to sensitive information by means of the data access pattern. Researches from MIT and the University of California argue that data stored in the cloud can be extracted and decrypted by other users of the service. Attackers could measure the memory of the access pattern which leaks sensitive information, causing a real threat to privacy (Islam *et al*., 2012).

Recent events have shown that online service providers are "curious" to acquire private information about users. Therefore, a general storage system is presented that hides data access patterns from the servers running it, thereby protecting user privacy. Early cloud storage spaces optimized traditional metrics such as performance, availability and scalability for distributed file systems. However, privacy has become an increasingly important affair. Mutually distrustful users expect the cloud provider to prevent unauthorized cross-user data access. However, users are entirely likely to distrust, in general, cloud providers themselves. Even if data is encrypted, user access patterns can leak important information (Troncoso *et al*., 2007; Kumar *et al*., 2007) pertaining to the content. Even without data access, an observer can see how often each item is accessed and then use statistical inferences to deduce the contents of encrypted items. For instance, over 80% of encrypted email queries are based on access patterns alone. A storage subsystem has been introduced that hides user access patterns from servers that store user data, implementing a distributed block interface (Lee and Thekkath, 1996) that allows clients to read and write to addresses 1-N where N is large. Block interface in oblivious storage servers cannot learn the plain text of user data, the addresses requested or the relationships between requested addresses.

MATERIALS AND METHODS

Bogdan proposed a mechanism that combines Oblivious RAM (O-RAM) access privacy and data confidentiality with Write-Once-Read-Many (WORM) regulatory data retention guarantees. Clients can with complete confidence, outsource their database to a server. Access privacy is ensured when client access patterns are tangibly hidden and the server cannot enforce access control directly. Client access is appended only when the data record has been written, since it cannot be removed or altered thereafter even by its writer. Williams *et al*. (2008) introduced a technique guaranteeing access pattern privacy against a computationally-bound adversary in outsourced data storage. Given the presence of a small quantity O (n) where n is the size of the temporary storage database, clients can achieve access pattern privacy with communication and computational

Corresponding Author: G. Sujitha, Rajalakshmi Engineering College, Anna University, 600025 Chennai, Tamil Nadu, India

complexities of less than O (log2n) per query. Pinkas and Reinman (2010) discovered an O-RAM construction which enables a client to store locally only a constant amount of data but to store remotely n data items and access them while hiding the identities of the items being accessed. Goodrich and Mitzenmacher (2010) proposed a novel O-RAM construction which achieves O ((log N)2) amortized cost with O (1) client-side storage or O (log N) amortized cost with O (N) client-side storage. Goodrich and Mitzenmacher (2010) construction achieves the best asymptotic performance of all known constructions. However, the running cost of sustaining their performance is prohibitive. Kushilevitz *et al.* (2012) introduced in the RAM model, the context of software protection by Goldreich and Ostrovsky (1996). A secure Oblivious RAM simulation allows for a client with a small (e.g., constant size) protected memory to hide not only data but also the sequence of locations, it accesses (both reads and writes) in the unprotected memory of size n. Carbunar and Sion (2011) proposed a mechanism for remote data storage with an efficient access pattern for privacy and correctness. A storage client can deploy this mechanism to issue encrypted reads, writes and inserts to a curious and malicious storage service provider, without revealing information or access patterns. The provider is unable to establish a correlation between successive accesses or even to distinguish between a read and a write. Chung proposed another statistically secure Binary-Tree O-RAM algorithm based on path ORAM. Their theoretical bandwidth bound is a log n factor worse than blocks of size (log N). Their simulation results suggest an empirical bucket size, meaning that their practical bandwidth cost is a constant factor worse than Path ORAM, since they require operating on 3 paths in expectation for each data access while Path ORAM requires reading and writing only 1 path.

Oblivious RAM was first investigated by Goldreich and Ostrovsky (1996), in the context of protecting outsourced data from piracy. The goal of O-RAM is to completely hide the data access pattern (which blocks were read/written) from the server. Each data read or write request will generate a completely random sequence of data accesses from the server's perspective. Shi *et al.* (2011) binary tree scheme is used to allow a client to conceal the data access pattern from remote storage spaces by continuously shuffling and re-encrypting data as they are accessed. An adversary can observe the physical storage locations accessed but the O-RAM algorithm ensures that the adversary has a negligible probability of learning anything about the true (logical) access pattern. In today's cloud era, clients wish to store data at a remote untrusted server while still preserving its privacy. While traditional encryption schemes can provide confidentiality, they do not hide the data access pattern which can reveal very sensitive information to the untrusted server, assuming that the server is untrusted and the client trusted. The goal is to completely hide the data access pattern with each data read or write request generating a completely random sequence of data accesses.

An oblivious RAM is a useful primitive for hiding the data access pattern and enabling privacy-preserving outsourced storage where clients store data at a remote, untrusted server. The O-RAM constructions (Goodrich and Mitzenmacher, 2010) inherit the hierarchical solution initially proposed by Goldreich and Ostrovsky (1996) and these constructions also inherit the periodic reshuffling operations required by the Goldreich and Ostrovsky (1996) construction. It can be used in conjunction with encryption to enable stronger privacy guarantees in outsourced storage applications.

The goal of an O-RAM is to completely obscure, the data access pattern (indicating which blocks were read/written) from the server. Data transfer is a major problem for client devices with a limited memory. An O-RAM is optimized by reducing the volume of data to be transferred per data access. These optimizations make an O-RAM most useful for personal storage in the cloud. The chief objective involves concealing the data access pattern in a tree structure and preserving the privacy of cloud data which poses a huge challenge.

The idea behind the working of an O-RAM is that each read or write generates a random sequence of accesses in the server, completely independent of the access pattern. Data transfer becomes the main bottleneck for O-RAMs well suited as, they are to client devices with limited memory such as smart phones and PDAs. Since, these devices are the primary computing platform for many users, progress is made by reducing as much as possible, the volume of data transferred on each request.

The challenge is to design an O-RAM that meets the following requirements: a client's storage requirements must be no more than a few megabytes. Clients with limited memory capacity may use storage as a service. Data owners hesitate to adopt the service if it requires large storage capacity on the client's side. Furthermore, the O-RAM's reshuffling process could take several hours if the data cannot fit into the memory available. Clearly such conditions are not practical if the client's is a handheld mobile device.

The amount of data transferred on each request must be relatively small enough to deliver good latency. A "pay-as-you-go" model user may be expected to tolerate a delay of no more than several seconds for a request.

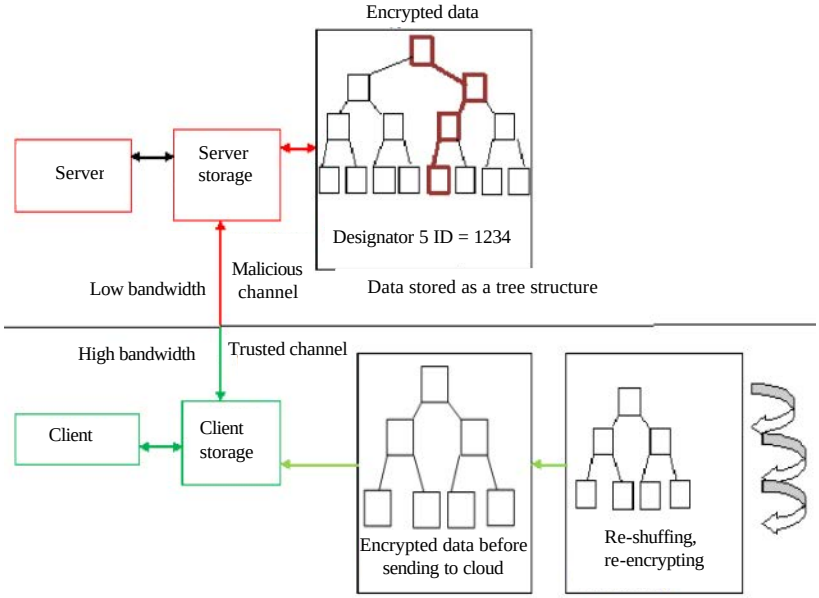

Fig. 1: Oblivious RAM architecture

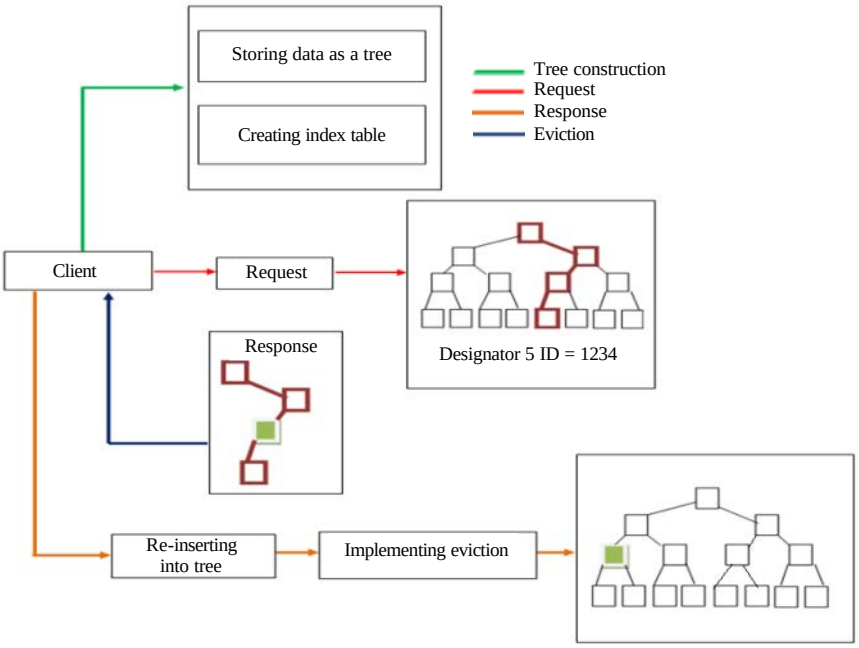

Fig. 2: O-RAM request-response

Oblivious RAM: Cloud servers are considered curious and untrusted entities. Data owners hesitate to adopt cloud technology if there are risks of data exposure to a third party or even cloud service providers themselves (Fig. 1 and 2). Therefore, providing adequate security and privacy protection for sensitive data is vital.

Statistical inferences of frequent queries submitted by clients allow providers to determine how frequently a particular piece of data is fetched from the cloud. Pinkas and Reinman (2010) gave an example in which a sequence of data access operations to specific locations (u1; 2; 3) can indicate a certain stock trading transaction and such financial information is often considered highly sensitive by organizations and individuals alike. The client retrieves all the blocks from the database to hide the patterns but there must be sufficient storage capacity on

the client's side to store data. If the client has low bandwidth, downloading the entire database may take up a lot of time. Further, since the cloud is a "pay-as you-go" model, the client needs to fork out substantial sums for the service, prompting them to think twice about cloud services in general.

System design: An oblivious RAM preserves privacy by continuously shuffling the location of the data as it is being accessed, thereby completely cloaking what data is being accessed or even when it was previously accessed. However, reshuffling could take hours if the data cannot fit into the memory available and such conditions are not feasible if the client has a handheld mobile device with limited bandwidth and storage capacity. For instance, a 64 Gbyte database consumes >200 Mbytes of user memory in the square root construction of Stefanov *et al.* (2011), a quantum of memory unavailable to most clients. The volume of data to be transferred per data access must, consequently be reduced.

O-RAM is based on binary-tree construction with storage organized into a binary tree over small data buckets. Data blocks are evicted in an oblivious fashion along tree edges from the root bucket to leaf buckets. In spirit, the binary-tree based construction tries to spread reshuffling costs over time; in reality, its operational mechanisms are based on Goldreich and Ostrovsky (1996)'s original hierarchical solution. Oblivious RAM construction leverages the available client-side storage as a working buffer and this allows drastic optimization of bandwidth consumption between server and client.

Let N denote O-RAM capacity (i.e., the maximum number of data blocks that an O-RAM can store) and assume that data is fetched and stored in atomic units called blocks. The client can read or write blocks. A read (v) operation takes an address v and returns the block with that address.

A write (v, b) operation takes an address v and a new block b and overwrites the old block at address v with b. Each operation is treated identically to make sure that the server cannot distinguish between reads and writes. This means ignoring the input block for a read and the output block for a write. Also, the client re-encrypts and rewrites each block that is accessed using a semantically secure encryption scheme so two encryptions of the same block are different, even if it has not been modified.

Hiding the data access pattern in an O-RAM tree structure: The O-RAM solutions use the basic memory structure suggested by Ostrovsky (1990)'s "Hierarchical Scheme", a structure arranged as a series of buckets. When a block is requested, the algorithm checks a bucket

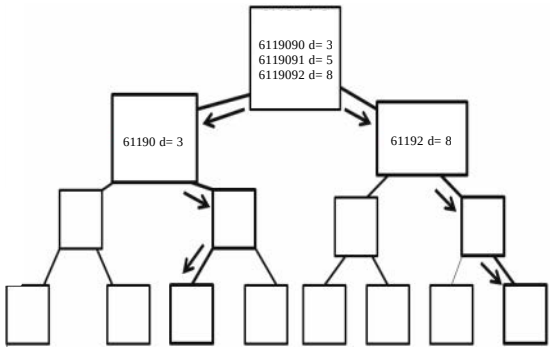

Fig. 3: O-RAM tree

Table I: Tree construction

No. of blocks in 2^k N	Bucket size log N	No. of paths for the block
2^2	2	4
2^3	3	8
2^4	4	16
2^5	5	32
2^6	6	64
2^7	7	128
2^8	8	256
2^9	9	512
2^10	10	1024

at each level of the hierarchy. If the block is found, the search continues to hide the location where the block was found. Finally, the block is reinserted into the top level and shuffled.

O-RAM binary tree: The O-RAM construction uses the binary tree scheme of Shi *et al.* (2011), the best known scheme for data requiring constant storage on the client's side. To store N data blocks, a binary tree of log N levels is initiated where each node is referred to as a bucket. Level i consists of up to 2^i buckets. Each bucket contains O (log N) entries, each of which comprises a unique id and a block (Fig. 3).

When a block is added to the tree, it is always inserted in the root bucket. Each block is labelled by a random number between 1 and N, called the designator, corresponding to the leaf bucket. The path from the root to the designator is the corresponding path towards which the block percolates down the tree (Table 1).

As each data block is logically assigned to a random leaf node every time it is operated on, there is a need for some kind of data structure to help remember where each block might be at a point in time. To track an entry's location in the tree, the client maintains a mapping from the unique id of the block to its designators in the index table as in Table 2. When data blocks are first added to the bucket at the root of the tree, the bucket's load increases as more data blocks continue to be added. To avoid the danger of the bucket's overflowing, data blocks

Table 2: Mapping of block to designator

Block Id	Designator
61190	7
62191	4
68178	8
61986	1
65431	3
67843	5
61658	6
62356	2

Table 3: Mapping of block to secret key

Block	Secret key
61190	I4NKoO+26JneOF5NIYAlHg==
61191	rOcYmleSO6rehJEoQqtg2Q==
61192	bS1DRw4KsitGjG4CDL+zQw==
N	-

residing in a non-leaf bucket are periodically evicted to its children buckets. Two blocks are randomly selected from the root bucket of the tree from a total of O (log N) entries in the bucket and evicted to prevent root buckets from overflowing. To evict from a bucket, a valid entry is removed from, it and added to the bucket's child along the path toward the evicted designator's leaf bucket. Each block gradually percolates down a path in the tree towards a leaf bucket until the block is read or written again.

Ensuring security every time a block is accessed is feasible when its designated leaf node is chosen independently and at random. This is necessary to ensure that two operations on the same data block are completely unlinkable.

Encryption: All data blocks are encrypted using a semantically secure encryption scheme before being inserted into the tree, so two encryptions of the same plain text cannot be linked. Furthermore, every time a data block is written back, it is re-encrypted again using a fresh private key. The client maintains a look-up table as in Table 3, mapping the block to its private key. To perform a read or write operation on the data block, it is decrypted by referring to the look-up table. Also, the client re-encrypts and rewrites each block that is accessed using a semantically secure encryption scheme so two encryptions of the same block are different-even if it has not been modified and the old private key of the block is replaced in the look-up table by the new key. This technique guarantees that the storage server cannot track repeated accesses to the same block.

RESULTS AND DISCUSSION

O-RAM operation: In an O-RAM, the database is considered a set of encrypted blocks and supported operations are read (block id) and write (block id, new value). Assuming that a block with address v starts at the root bucket and slowly percolates down the tree toward the leaf indicated by the designator, the block will always be found somewhere along the path from the root to the corresponding designator. Then, a read or write operation can be performed by reading all buckets in the tree along the path indicated by the designator. When, the block is found, it is removed from its current bucket and added to the root bucket using a new designator. Every time, the client requests a read or write at an address v, the corresponding block is assigned a new random leaf bucket and reinserted in the root bucket. It follows that the paths taken to look up v in two different requests are independent of one another and cannot be distinguished from the lookup of any other two addresses.

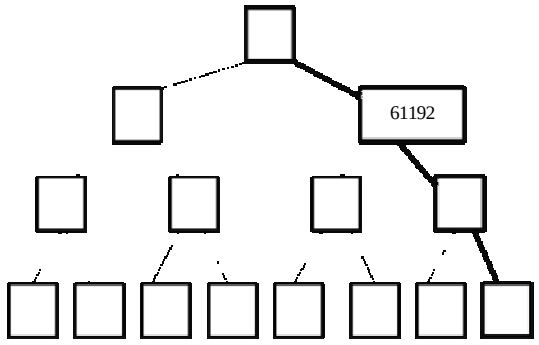

Fig. 4: Search for a data block

Read and write operations: The standard O-RAM adopted (Sanchez-Artigas, 2013) (worst) exports read and write interfaces. To hide whether the operation is a read or a write either operation generates both a read and a write to the O-RAM. Goldreich and Ostrovsky (1996) proposed O-RAMs that support a few enriched operations, exporting a read and write operation and a re-insertion operation. The performance of searching for a block is described in Fig. 4. The client looks up the index table to find out which designator is associated with the requested block id. For instance, in the case of read (61192), index structure Table 2 is referred to for the designator associated with the block id 61192.

Upon receiving all the buckets in the corresponding path, the client decrypts the necessary block using the look-up Table 3. Data remains unchanged, the operation being read. After the read operation, the client chooses a new designator (new random leaf) for the block and updates the index table with the new designator value Table 4.

Even when, the same block is requested the next time, it is fetched from the path that is different from the previous access, ensuring that there is no linkability

Table 4: Updated index table

Block	Designator
61190	5
61191	4
61192	8
N	-

Table 5: Updated secret key look-up table

Points	Secret key
1	I4NKoO+26JneOF5NIYAlHg==
2	rOcYmleSO6rehJEoQqtg2Q==
3	bS1DRw4KsitGjG4CDL+zQw=

between two operations on the same data block. The client then removes the block from the bucket where it was found and puts it instead in the root bucket in Fig. 2. Before being inserted into the root bucket, the block is re-encrypted and the new secret key replaced in the look-up (Table 5), so two encryptions of the same block are different even if it has not been modified. This technique guarantees that the storage server cannot track repeated accesses to the same block. Since, the new re-encrypted block is placed at the root, this operation does not violate the tree invariant that is being maintained. Every time, the data block is accessed from the O-RAM tree, its designator is changed randomly and updated in the index table. Even if queries submitted by the client access the same data block each time, it is fetched from a different path.

The cloud provider can have no access to sensitive information about data blocks by using access patterns, since patterns change with each request.

A statistical inference of frequent queries submitted by the client does not allow the provider to determine how frequently a particular piece of data is fetched from the cloud. This is because data is re-encrypted and decrypted each time using a different secret key at each request. When a client requests a write operation at a block, the said client looks up the block's designator in a table that maps the block id to its designators. There after, all buckets in the tree along the path between the root and its designator are read. When the entry is found, it is removed from its current bucket and rewritten at the top of the tree using a new designator. Before being inserted into the root bucket, it is re-encrypted and the secret key look-up table is updated. Thus, repeated reads for the same entry produce different lookup paths through the tree.

Table 5, mapping block id to designators and block id to the secret key, must itself be accessed obliviously. However, it contains O (N) mappings, making it far too large to be stored locally. Fortunately, each mapping is tiny, no more than small bits. Thus, recursively, the same O-RAM scheme can also be applied to a smaller collection of blocks.

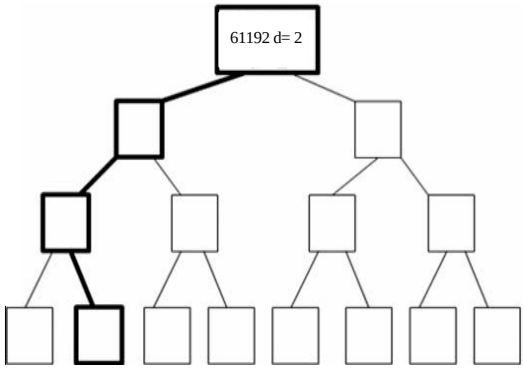

Fig. 5: Tree re-insertion

Optimizing oblivious-RAM: An oblivious RAM allows a client to outsource her data to a remote server (e.g., the cloud) and access, it efficiently and privately. In particular, the client can access individual elements of her data without disclosing to the server the elements, she is accessing. The quantum of time spent by the client and server on each such data access should be small and essentially independent of the size of the client's entire data. Using an O-RAM, a client can privately execute arbitrary RAM computations over her remotely stored data without having to download the data from the server in its entirety. In particular, the client/server computation and communication are essentially only proportional to the time-complexity of the RAM computation itself. Therefore, O-RAM offers tremendous savings when the client wants to execute simple computations (e.g., binary search) over huge amounts of data.

Binary tree: To store N data blocks, a binary tree of log N levels is initiated where each node is referred to as a bucket. Level i consists of up to 2^i buckets. Each bucket contains O (log N) entries, each of which contains a unique id and a block (Fig. 5). To find the entry for a given id, the index table is referred to for the block's designator and the path through the tree followed, dictated by that designator. For each bucket on the path, all entries are read in that bucket. This continues along the entire path, even if the block being looked for is found, so as to hide the entry's actual location from the server. After reading or writing the entry, we remove it from wherever it was found, assign it a new designator, re-encrypt and reinsert it into the top-level bucket. Since, the block-to-designator table is large, it is itself stored in a recursive version of the O-RAM structure. Each stage is smaller than the one it stores designators for.

Bucket structure: Data blocks are evicted in an oblivious fashion along tree edges from the root bucket to leaf

buckets. In spirit, the binary-tree based construction tries to spread reshuffling costs over time while, in reality, its operational mechanisms bear little resemblance to prior schemes (worst) based on Goldreich and Ostrovsky (1996)'s original hierarchical solution (worst). As entries keep getting added to the root bucket, it eventually overflows. To prevent internal buckets from overflowing, the client must evict-on each request-blocks from internal buckets to their children. While the server learns which buckets were chosen for eviction, the eviction algorithm makes sure that the server does not know which child buckets received the evicted blocks. If no empty space exists in that child, the eviction algorithm fails. The size of buckets has a determining impact on the efficiency of the scheme. Since, bucket size is largely determined by the efficacy of the eviction procedure, a natural idea to reduce the capacity of buckets is to improve eviction.

Optimization: At each node of the tree, a bucket has the capacity for several blocks. The problem lies in that bucket size has a strong impact on the scheme's efficiency. The reason is that the client needs to search over L log N items for buckets of size L. If the buckets are too small, there is every probability of their overflowing and any overflow can leak information about access patterns. For example, let us suppose that a bucket near the root of a tree overflows. The cloud provider realizes that certain of the last-accessed blocks lie in that part of the tree and, consequently can likely learn something about private data based on the pattern of immediate accesses to the tree. Therefore, it is necessary to choose bucket size for efficiency with bucket size reduced to log N where N is the database size. If N = 1024 then bucket size is 10 which could be prohibitive for client devices.

Eviction procedure: Upon every data access operation for each depth in the hierarchy, numbers of buckets are chosen randomly for eviction during which one arbitrary data block is evicted to each of its children. More specifically, eviction is an oblivious protocol between client and server in which the client reads data blocks from selected buckets and writes each block to a child bucket. Over time, each block gradually percolates down a path in the tree towards a leaf bucket, until the block is read or written again. Whenever a block is being added to the root bucket, it is logically assigned to a random leaf bucket, thereby maintaining the index table. Thereafter, this data block gradually percolates down towards the designated leaf bucket, until the same data block is read or written again. To find the particular data block, it is essential to search the data block in all buckets on the path from the designated leaf node to the root (Fig. 6).

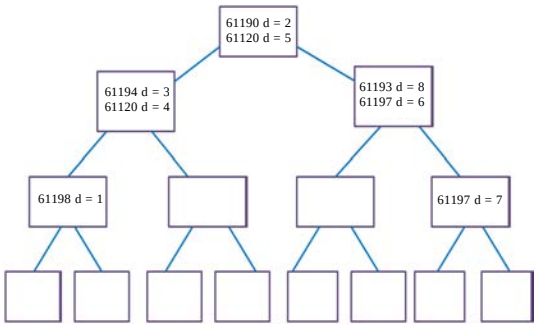

Fig. 6: Tree before implementing eviction

For security reasons, it is important to ensure that every time a block is accessed, its designated leaf node must be chosen independently and at random. This is necessary to ensure that two operations on the same data block are completely unlinkable. The bucket sequence accessed during the eviction process must reveal no information about the load of each bucket or the data access sequence. The choice of which buckets are to be evicted is randomly selected and independent from the load of the bucket or the data access sequence.

Eviction rate: The client must, on each request, evict blocks from internal buckets to their children. At each level of the tree, the client randomly chooses buckets for eviction. Let, the eviction rate be two. Whenever, the eviction algorithm is invoked, the client randomly selects two buckets to evict along every depth of the tree. To evict from a bucket, an arbitrary data block is removed from the selected bucket and inserted in the child bucket lying on the path toward the designated leaf node. If no empty space exists in that child then two arbitrary data blocks are removed from that particular child bucket and inserted into the bucket below, it that lies on the path toward the designated leaf node (Fig. 7).

Eviction is now performed in the child bucket. Finally, optimizing construction when searching for a data block, the client accesses every bucket on the path from the root to the leaf indicated by the designator. There is then, a natural optimization to take advantage of each path traversal so as to opportunistically move blocks down the tree. From the standpoint of privacy, nothing prevents the client from moving blocks from the currently accessed bucket to its child in the path. The client has already scanned this bucket during path traversal; therefore the sequence of accessed buckets is identical to that of the original construction. The only requirement is for the client to have sufficient memory to store two buckets: the current bucket and its child which is less than the storage required by our new eviction procedure and algorithm (Fig. 8-10).

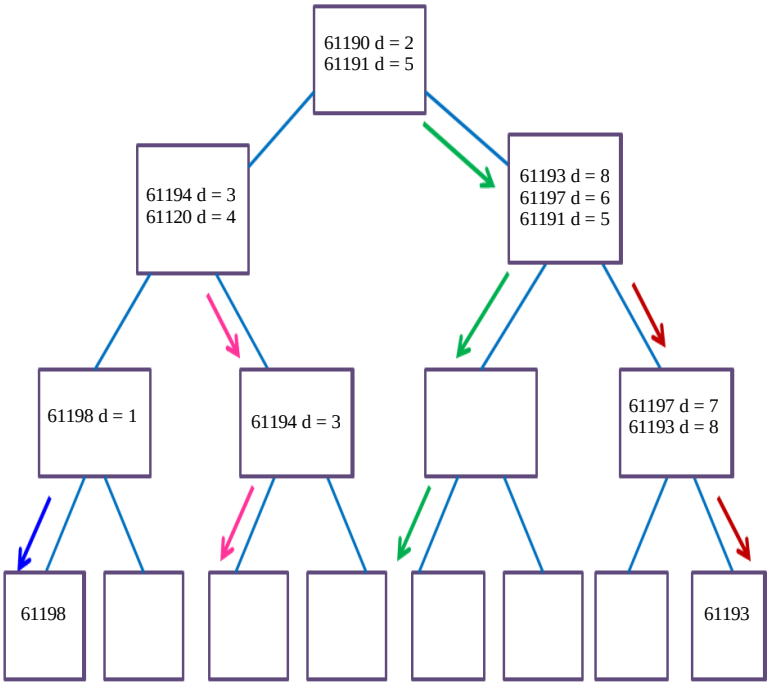

Fig. 7: Tree after implementing eviction

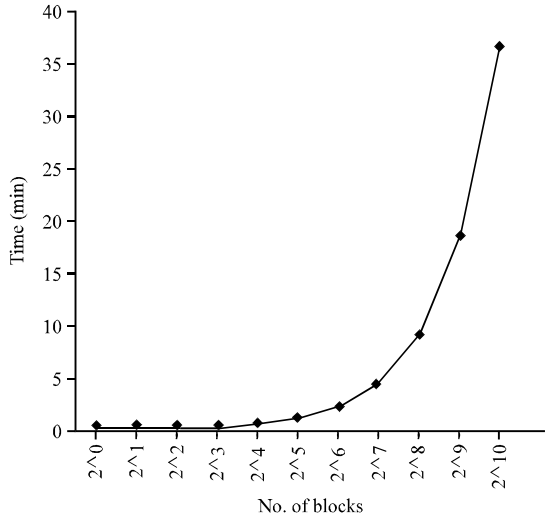

Fig. 8: O-RAM tree construction time

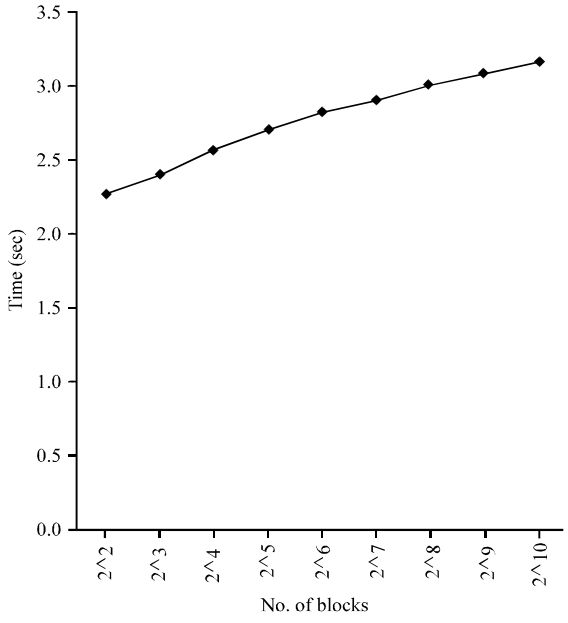

Fig. 9: Eviction time

Eviction algorithm

Eviction:

```
For d = D-1 to 0 do */D is the number of levels*/
Compute the set of the buckets to evict
at level D
for each bucket in Buckets to evicit do
        for each block in buckets do
        1←designator of block
        b←(D-d-1) th bit of
        if child_b is not full then
        Child_b. Add (block)
        end if
        end for
end for
end for
```

Let, L be the bucket capacity and l the size of a block in bits. Observing that a read or write request performs

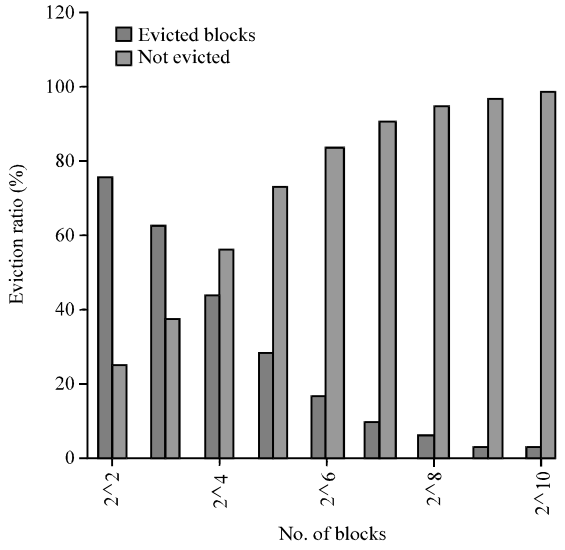

Fig. 10: Evicted ratio

one operation on each of log N buckets, the amount of data transferred thus is lL log N. Similarly, the eviction algorithm operates on O (log N) buckets (two buckets per level). Then, the cost is O (lL log N). Setting L = O (log N) to prevent bucket overflows, the overall worst case complexity of this scheme is O (l log2 N).

Evaluation: To assess the practicality of O-RAM for different values of N where N is the db size, tree construction and O-RAM operations are evaluated. Bucket size depends on the size of N, with smaller buckets being used for efficient eviction procedures. If the buckets are too small, there is a high probability of their overflowing and any overflow can leak information about the access pattern. Bucket size is reduced to log N. The key observation is that as N increases, the relative bucket size of our scheme decreases significantly and tree construction time increases.

Eviction is performed for each read or write request after being re-inserted into the tree. At each level of the tree, the client randomly chooses two buckets and one arbitrary data block. The tree level increases as N increases and the time taken for eviction also increases.

Oblivious RAM construction is efficient as database size increases. In general, a reshuffle could take hours for a large dataset. In the O-RAM storage system, since the client randomly selects from the chosen bucket only two buckets and one arbitrary block for eviction, the number of evicted blocks decreases as N increases.

CONCLUSION

Securing user data within a data center requires more than mere encryption for hiding data access patterns adds additional protection against malicious servers and hackers. An oblivious RAM hides all information about block accesses. Deploying such oblivious storage in a data center creates new challenges and opportunities, including issues of scale, parallelism, maliciousness, fault tolerance and worst-case performances. Data transfer becomes the main bottleneck for O-RAMs well suited as they are to client devices with limited memory such as smart phones and PDAs. Since, these devices are the primary computing platform for many users, we have made progress in reducing as much as possible, the volume of data transferred on each request. These optimizations are key to make O-RAMs attractive for personal storage.

IMPLEMENTATIONS

Future enhancements of this research include implementing a path oblivious RAM, an efficient integrity verification method (Ren *et al*., 2013) and improving integrity verification to increase Path ORAM latency by 17%.

REFERENCES

Carbunar, B. and R. Sion, 2011. Write-once read-many oblivious RAM. IEEE Trans. Inform. Forensics Secur., 6: 1394-1403.

Chor, B., E. Kushilevitz, O. Goldreich and M. Sudan, 1998. Private information retrieval. J. ACM, 45: 965-981.

Goldreich, O. and R. Ostrovsky, 1996. Software protection and simulation on oblivious RAMs. J. ACM., 43: 431-473.

Goodrich, M.T. and M. Mitzenmacher, 2010. Mapreduce parallel cuckoo hashing and oblivious ram simulations. https://arxiv.org/pdf/1007.1259.pdf.

Islam, M.S., M. Kuzu and M. Kantarcioglu, 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. Proceedings of the 19th Annual Network and Distributed System Security Symposium, February 5-8, 2012, San Diego, CA., USA., pp: 1-14.

Kumar, R., J. Novak, B. Pang and A. Tomkins, 2007. On anonymizing query logs via token-based hashing. Proceedings of the 16th International Conference on World Wide Web, May 8-12, 2007, Banff, AB, Canada, pp: 629-638.

Kushilevitz, E., S. Lu and R. Ostrovsky, 2012. On the (in)security of hash-based oblivious RAM and a new balancing scheme. Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, January 17-19, 2012, Kyoto, Japan, pp: 143-156.

Lee, E.K. and C.A. Thekkath, 1996. Petal: Distributed virtual disks. ACM SIGPLAN Notices, 31: 84-92.

Ostrovsky, R., 1990. Efficient computation on oblivious RAMs. Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, MD., USA., pp: 514-523.

Pinkas, B. and T. Reinman, 2010. Oblivious RAM revisited. Proceedings of the 30th Annual Cryptology Conference, August 15-19, 2010, Santa Barbara, CA., USA., pp: 502-519.

Ren, L., C.W. Fletcher, X. Yu, M. van Dijk and S. Devadas, 2013. Integrity verification for path oblivious-RAM. Proceedings of the 17th IEEE High Performance Extreme Computing Conference, September 10-12, 2013, Waltham, MA., USA., pp: 1-6.

Sanchez-Artigas, M., 2013. Toward efficient data access privacy in the cloud. IEEE Commun. Mag., 51: 39-45.

Shi, E., T.H.H. Chan, E. Stefanov and M. Li, 2011. Oblivious RAM with O((logN)3) worst-case cost. Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security, December 4-8, 2011, Seoul, South Korea, pp: 197-214.

Stefanov, E., E. Shi and D. Song, 2011. Towards practical oblivious RAM. http://arxiv.org/pdf/1106.3652.pdf.

Troncoso, C., C. Diaz, O. Dunkelman and B. Preneel, 2007. Traffic analysis attacks on a continuously- observable steganographic file system. Proceedings of the 9th International Workshop on Information Hiding, June 11-13, 2007, Saint Malo, France, pp: 220-236.

Williams, P., R. Sion and B. Carbunar, 2008. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. Proceedings of the 15th ACM Conference on Computer and Communications Security, October 27-31, 2008, Alexandria, VA., USA., pp: 139-148.