Oblivious RAM Revisited

To appear in Crypto 2010

Benny Pinkas

Tzachy Reinman

Problem Statement

Client with small secure memory. Untrusted server with large storage.

Server farm. Cloud storage.

Capacity: n data items

Client

Capacity:
O(1) data items
log(n) bit counter

Client can store data with the server

- Can encrypt data to hide its contents
- Can MAC data to prevent server from changing it

Still, the server can track the client's access pattern

- Learn which data items are accessed more frequently
- Relate between access patterns and auxiliary knowledge , such as stock-exchange action

We would therefore like the access pattern to be oblivious:

- For any two equal length sequences y,y' of R/W operations, the views of the server must be computationally indistinguishable
- I.e., the server does not know whether client accesses items (1,2,3,4) or items (1,2,2,1)

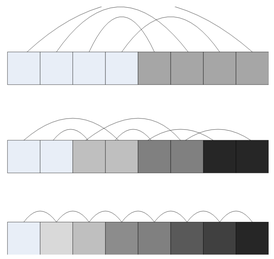
Assumptions and Requirements

- Client must have a private source of randomness
- Data must be encrypted with a semantically secure encryption scheme
- Each access to remote storage must include a read and a write operations
- Client stores n data items, of equal size, of the form ($index_i$, $data\text{-}block_i$) $\forall i,j$ $index_i \neq index_j$
- The location in which data item ($index_i$, $data\text{-}block_i$) is stored must be independent of $index_i$
- Two accesses to $index_i$ must not necessarily access the same location of the remote storage

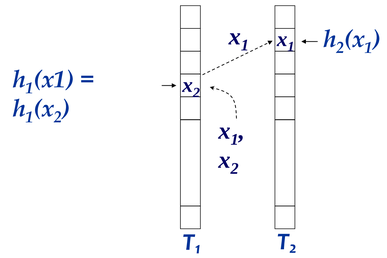
Comparison to Previous Work

	Computational Overhead	Client memory	Server memory
Simple [GO]	O(√n·logn)	O(1)	O(n+√n)
Hierarchical (Batcher) [GO]	$O(\log^4 n)$	O(1)	O(n·logn)
Hierarchical (AKS) [GO]	$O(\log^3 n)$ const >6100	O(1)	O(n·logn)
Merge sort [WS08]	$O(\log^2 n)$	O(√n)	O(n·logn)
Bloom filter [WSC09]	O(logn·loglogn) additional BF const > 92	O(√n)	O(n)
Ours	**$O(\log^2 n)$**	**O(1)**	**O(n)**

Tools



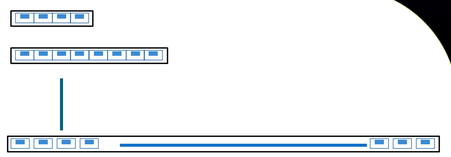
Randomized Shell Sort [G]
Oblivious sorting takes only O(n·logn)



Cuckoo Hashing [PR]
Requires only O(n) storage

Our Solution

Hierarchical solution
(based on [GO])



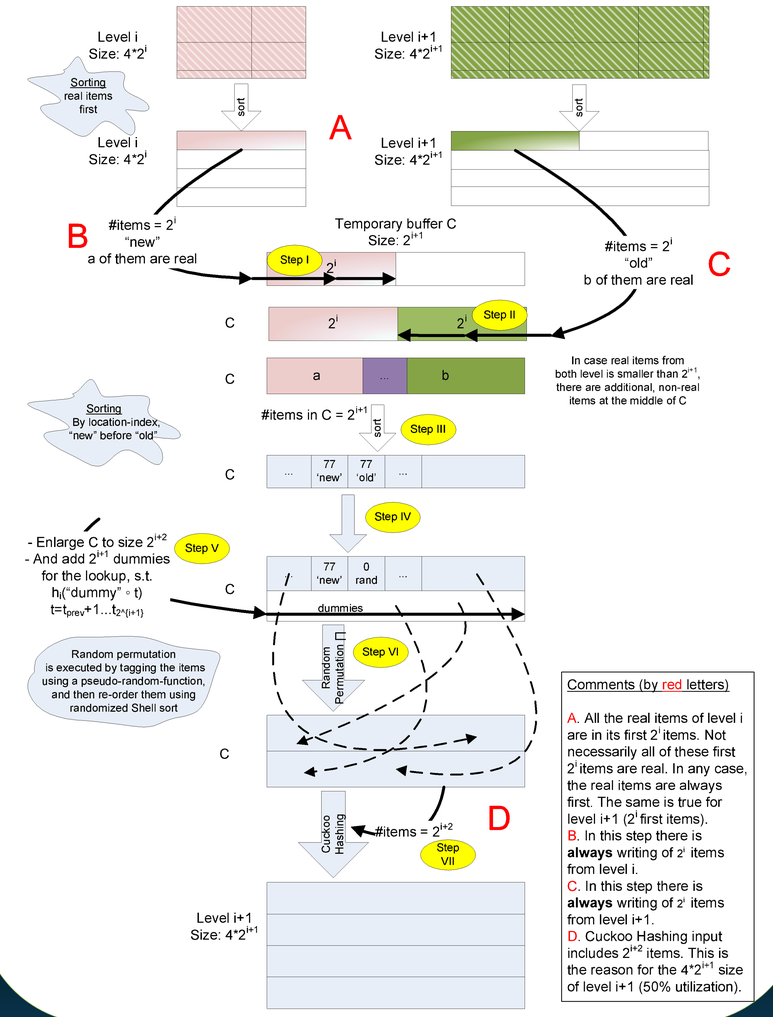
The server must not know in which level the accessed item is found

- log n levels, level i stores $4 \cdot 2^i$ items
- Each level is associated with two random hash functions
- To access an item –
  - Search the entire first level
  - For each level other than the first – examine two locations according to the level's hash function
    - If the item has not been found yet, the input to the hash function is the index, otherwise – it is random
  - Go over the entire first level and write the value to the next available location

Reshuffle Levels

After 2^i R/W ops, level i becomes full and is obliviously reshuffled into level i+1, using Randomized Shell sort and Cuckoo hashing with two new random secret hash functions.

The resulting level i+1 is ordered independently of any of the levels before the reshuffling.



Acknowledgments

Thanks to Yuriy Arbitman for informing us of the randomized Shell sort paper.

References

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

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

[PR] Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms 51(2), 122-144 (2004)

[WS08] Williams, P., Sion, R.: Usable PIR. In: NDSS (2008)

[WSC09] Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: ACM Conference on Computer and Communications Security. pp. 139-148 (2008)