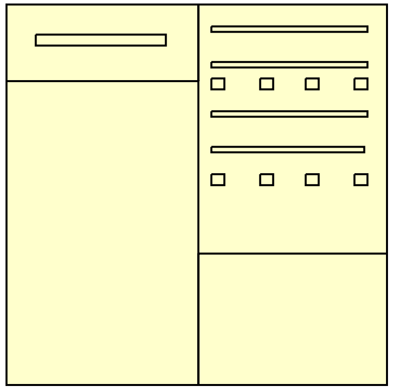
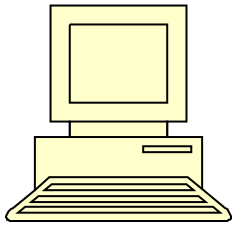
Oblivious RAM

Benny Pinkas

Bar-Ilan University

Oblivious RAM – the setting

- Setting: Client with small secure memory. Untrusted server with large storage.

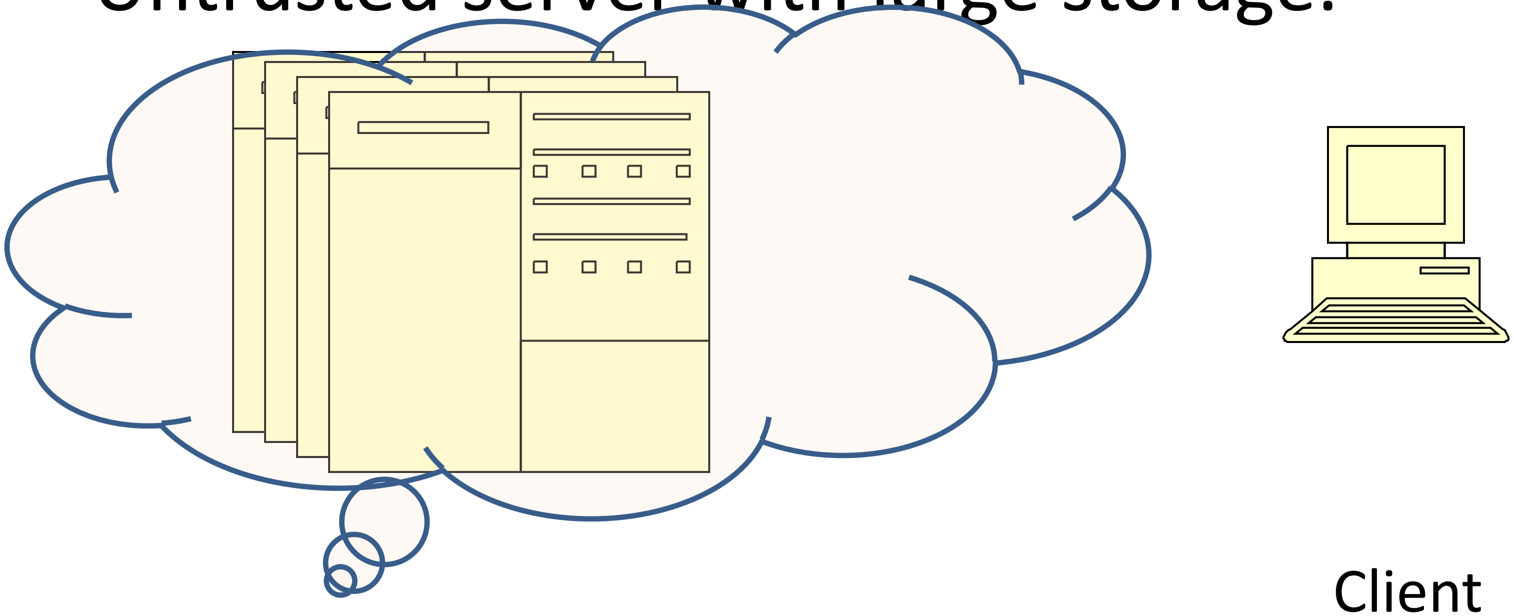



server

client

Oblivious RAM – the setting

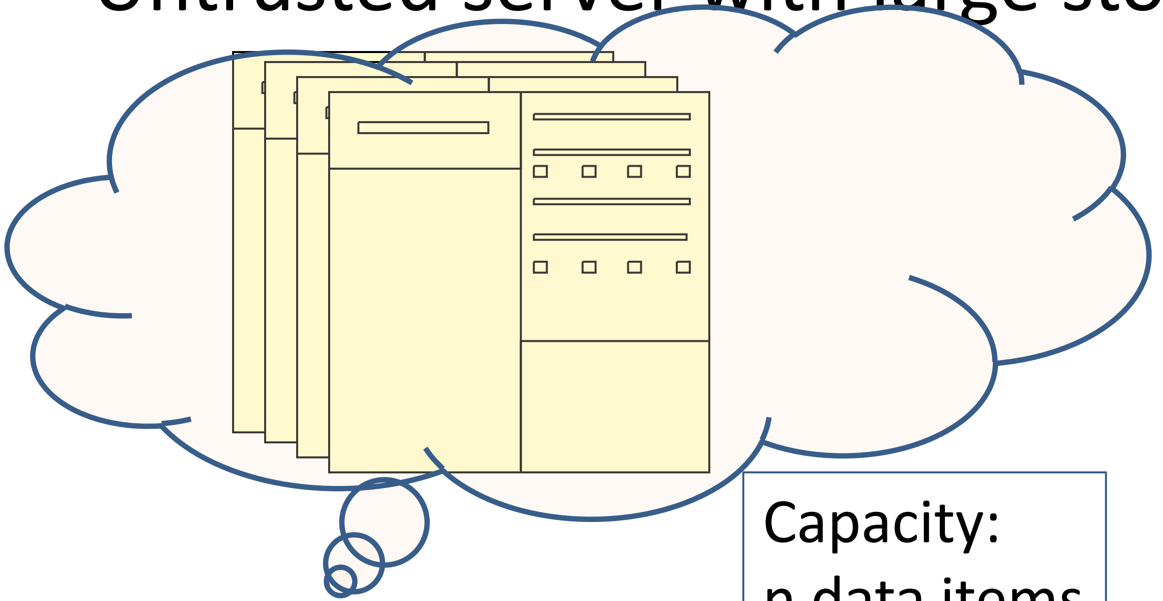
- Setting: Client with small secure memory. Untrusted server with large storage.



Server farm
Cloud storage

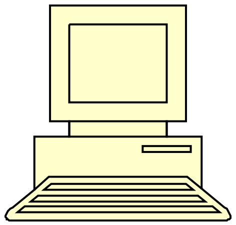
Oblivious RAM – the setting

- Setting: Client with small secure memory. Untrusted server with large storage.



Capacity:
n data items

Server farm
Cloud storage

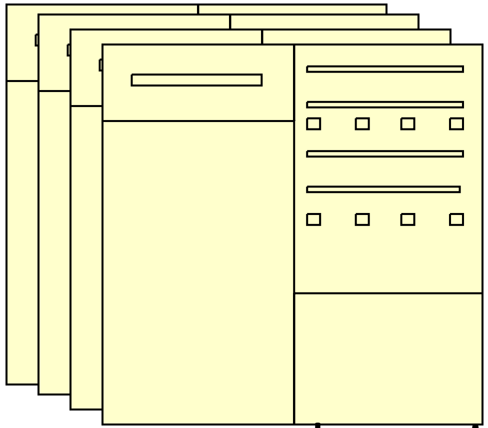


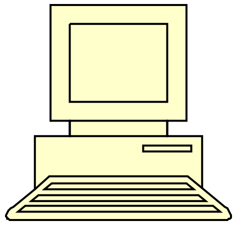
Capacity:
O(1) data items
log(n) bit *counter*

Client

Oblivious RAM – the setting

- Setting: Client with small secure memory. Untrusted server with large storage.





- Client can store data with the server
  - Can encrypt data to hide its contents
  - MAC data to prevent server from changing it
  - **But also desires to hide access pattern to data**

Oblivious RAM – the setting

Hiding access pattern to data: Server does not know whether client access the items numbered (1,2,3,4) or items (1,2,2,1)

- Client can store data with the server
  - Can encrypt data to hide its contents
  - MAC data to prevent server from changing it
  - **But also desires to hide access pattern to data**

Oblivious RAM - definition

- Client
  - Stores n data items, of equal size, of the form $(index_i, data\ block_i)$. $\forall i,j\ \ index_i \neq\ \ indexj$
  - Performs a sequence y of n read/write ops
- Access pattern $A(y)$ to remote storage contains
  - Remote storage indices accessed
  - Data read and written
- Secure oblivious RAM: for any two sequences y,y' of equal length, access patterns A(y) and A(y') are computationally indistinguishable.

Immediate implications of ORAM Definition

- Client must have a private source of randomness
- Data must be encrypted with a semantically secure encryption scheme
- Each access to the remote storage must include a read and a write
- The *location* in which data item *(index$_i$, datablock$_i$)* is stored must be independent of *index$_i$*
- **Two accesses to *index$_i$* must not necessarily access the same location of the remote storage**

Oblivious RAM - applications

- Related to Pippenger and Fischer’s 1979 result on oblivious simulation of Turing machines
- Software protection (Goldreich Ostrovsky)
  - CPU = client, RAM = remote storage
  - Prevent reverse engineering of programs
- Remote storage (in the “cloud”)
- Search on encrypted data
- Preventing cache attacks (Osvik-Shamir-Tromer)
- Secure computation

Trivial solution

- **For every R/W operation**
  - Client reads entire storage, item by item
  - Re-encrypts each item after possibly changing it
  - Writes the item back to remote storage
- **O(n) overhead per each R/W operation**

The Goldreich-Ostrovsky Constructions

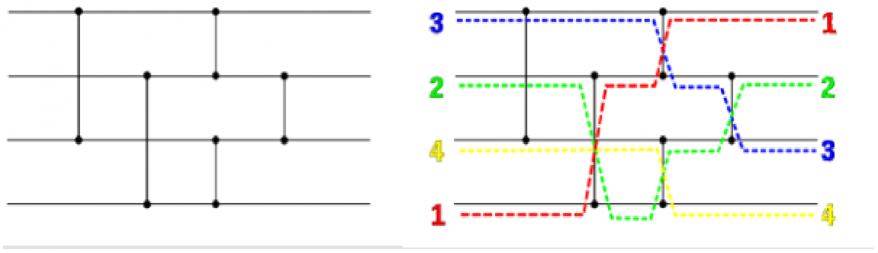
Software protection and simulation on oblivious RAMs, O. Goldreich and R. Ostrovsky, *Journal of the ACM (JACM)* 43, no. 3 (1996): 431-473.

Basic Tool: Oblivious Sort

- The client has stored n encrypted items on a remote server.
- The client needs to **obliviously** sort the items according to some key.
  - Comparing two items can be done by downloading them to the client, decrypting and comparing them.
  - But the server is aware which items the client downloads.

Oblivious Sort

- **Oblivious sort: the sequence of comparisons is independent of the input**
  - Naïve Bubble Sort √ (but $O(n^2)$)
  - Quick Sort O(nlogn) X
  - Sorting network √
    - Basic primitive – **black box comparator**
    - Batcher - $O(n\log^2 n)$
    - AKS - O(nlogn), but > 6100·nlogn

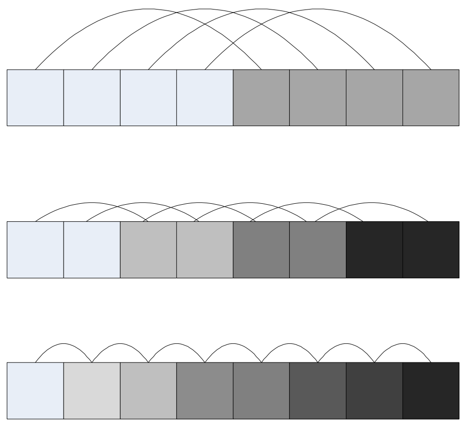


- Randomized Shell sort…

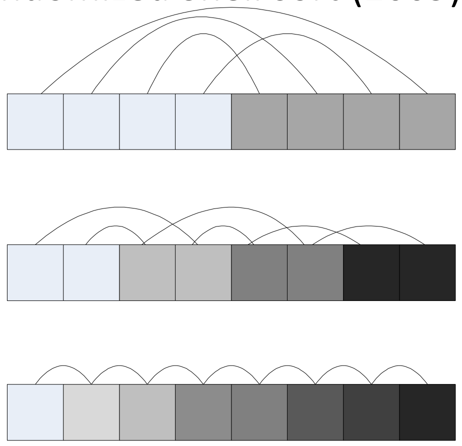
Randomized Shell sort

- Goodrich (2009)
- A randomized version of Shell sort (Shell, 1959).
- Oblivious - The distribution of the items that are compared is independent of their values.
- Efficient - O(nlogn), small constant, correct whp.

Shell sort (1959)

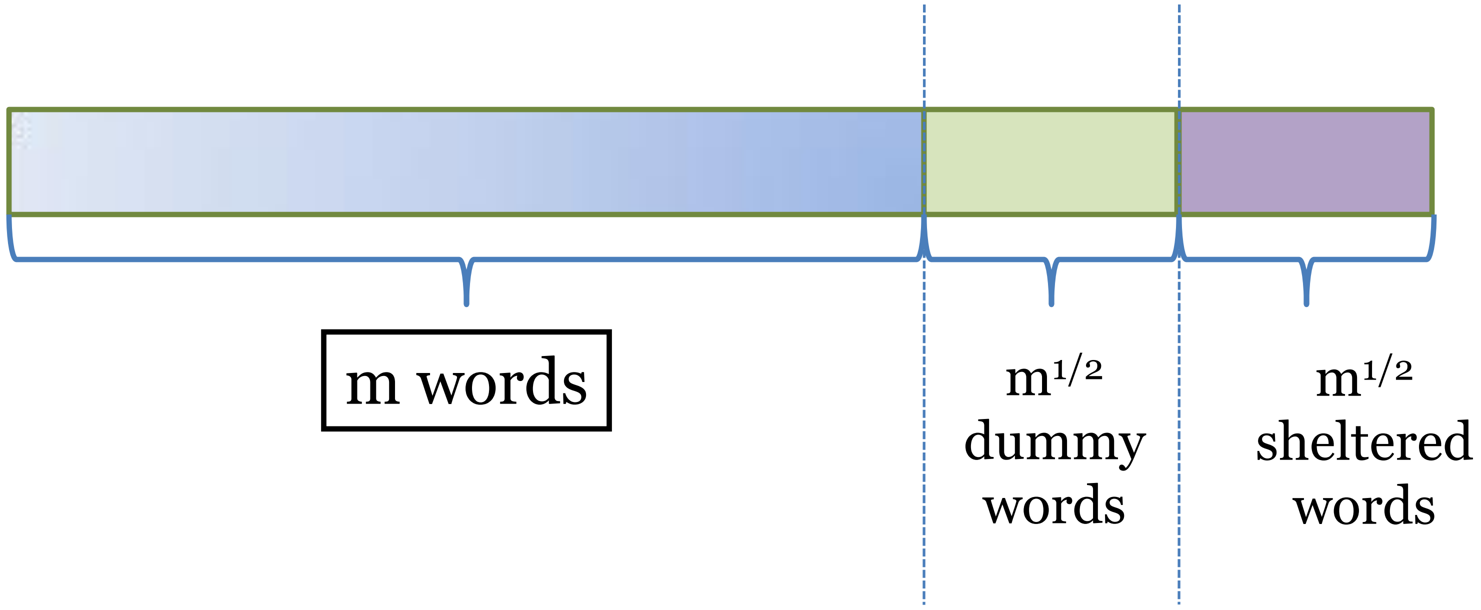


Randomized Shell sort (2009)



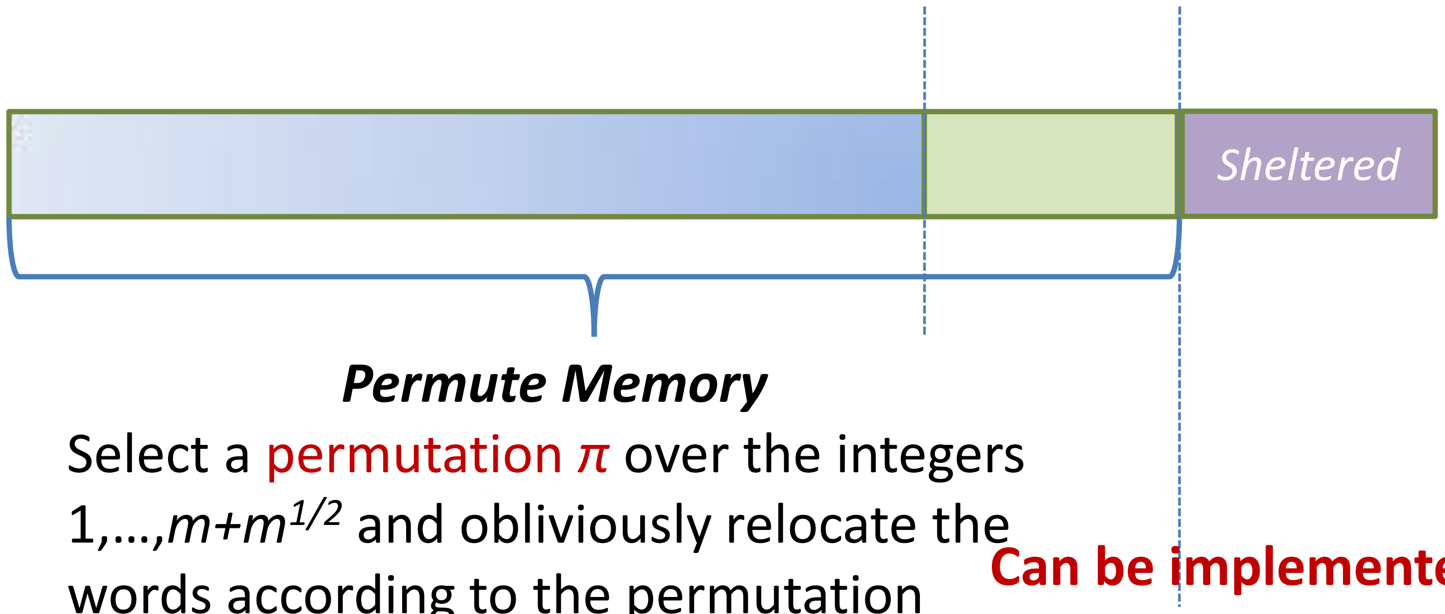
Square Root ORAM

- **First Step (once),** ***Initialization***



Square Root ORAM

- Second Step



Permute Memory

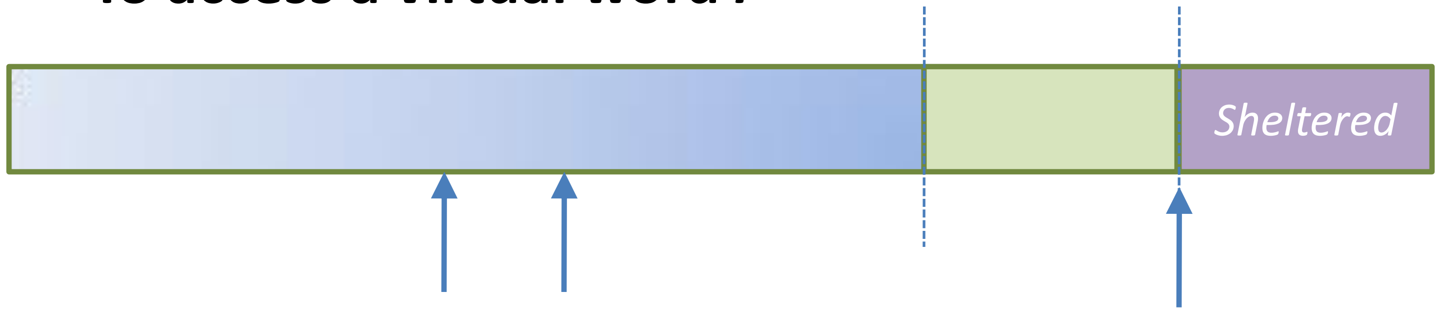
Select a permutation π over the integers $1,\ldots,m+m^{1/2}$ and obliviously relocate the words according to the permutation

Can be implemented using oblivious sort

Square Root ORAM

- ***Accessing the RAM***

  To access a virtual word *i*



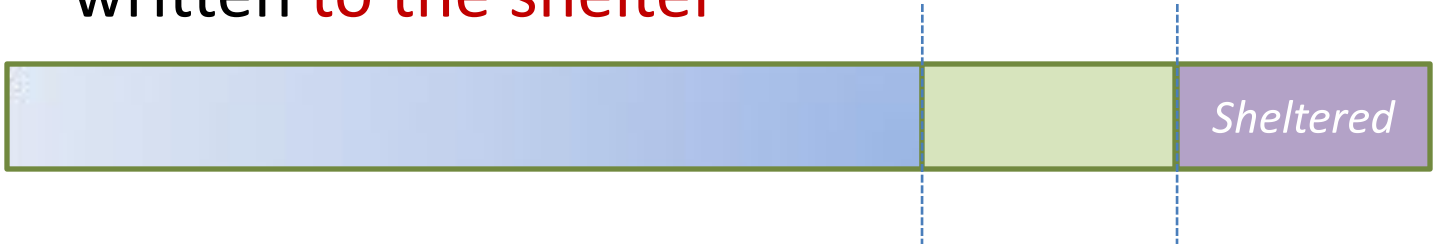
If *not found* in the shelter go to the actual word *π(i)*

If *found* in the shelter, access the next dummy (in the actual address *π(m+j)* where *j* is the step# in this epoch)

Scan through the entire shelter in a predefined order

Square Root ORAM

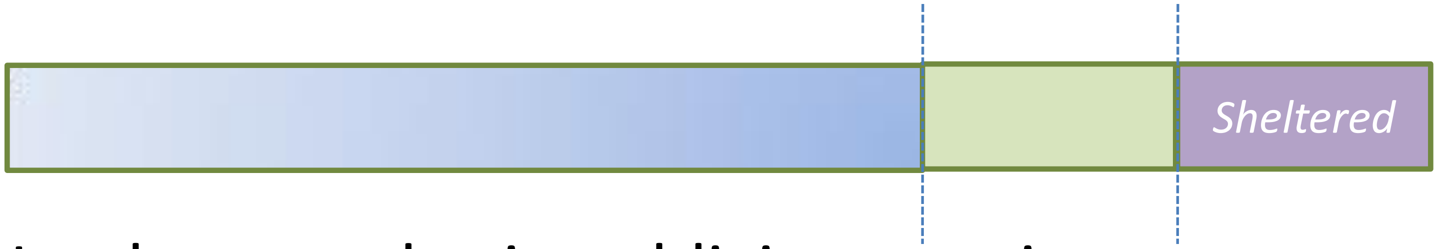
- ***Writing to the Shelter***
  The updated value for the i^{th} virtual location is written to the shelter



- update is done *IN ANY CASE*, and it is done by scanning ALL the shelter
- Obviously, after $m^{1/2}$ I/O ops, shelter becomes full

Square Root ORAM

- ***Updating the permuted memory***
  After $m^{1/2}$ accesses, the shelter values obliviously update the content of the permuted memory



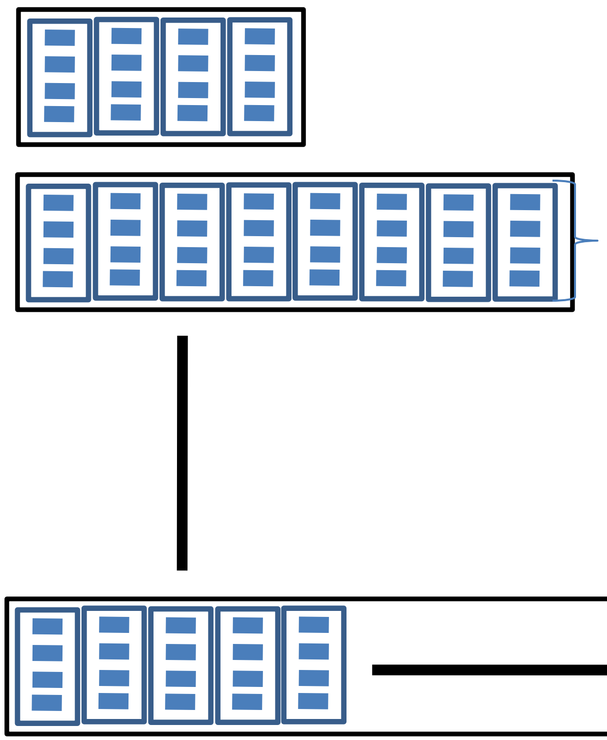
- Implemented using oblivious sorting…
- The overhead using Batcher network is $O(m\log^2(m))$

Square Root ORAM

- Overhead:
  - Each access requires reading all $m^{1/2}$ sheltered items
  - After $m^{1/2}$ accesses, must sort all items at a cost of $O(m\log^2(m))$
  - Overall an amortized cost of $O(m^{1/2}\log^2(m))$
  - Relatively simple to implement. No hidden constants.
- Security?

The Hierarchical Solution [GO]

Level *i* stores at most 2^i items

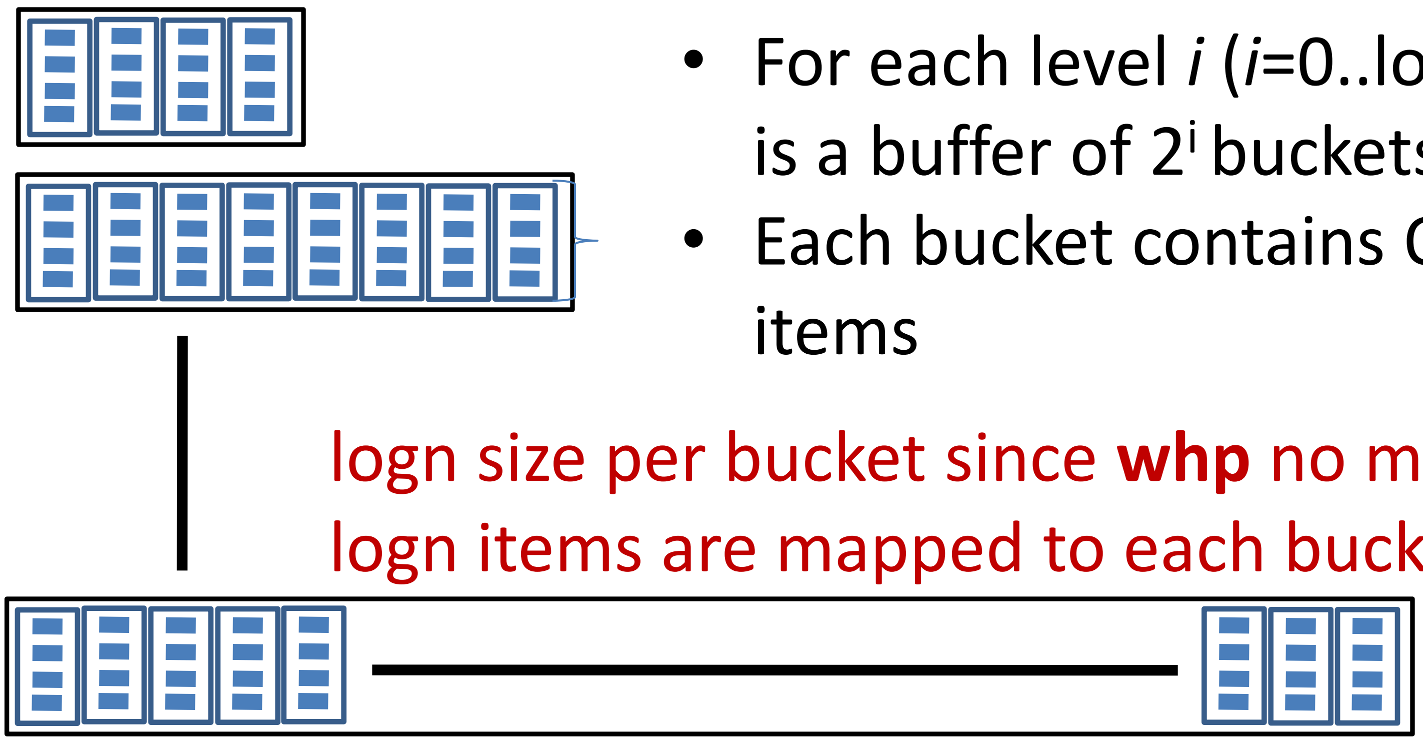


- Data items are arranged in logn levels
- For each level *i* (*i*=0..log t) there is a buffer of 2^i buckets
- Each bucket contains O(log n) items
- Invariant: item x is located in one of the levels, in bucket $H_i(x)$

($H_i()$ maps values to one of the 2^i buckets in level i)

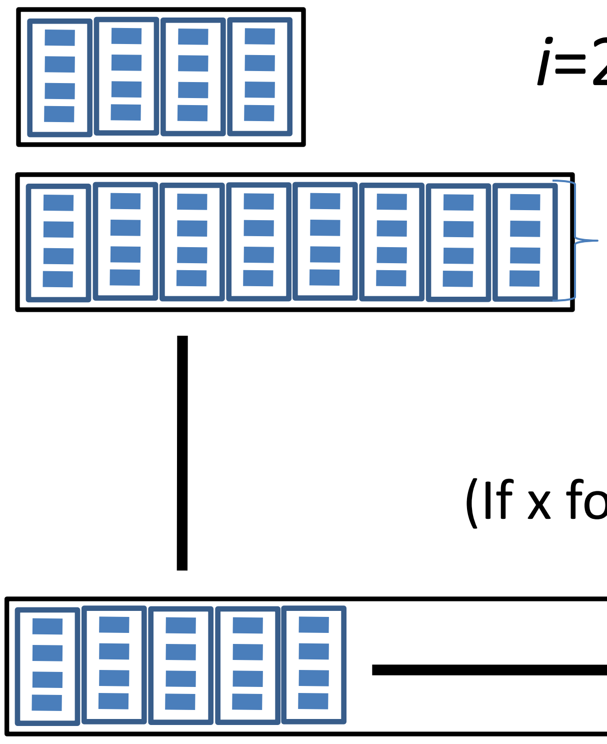
The Hierarchical Solution [GO]

- Data items are arranged in logn levels
- For each level *i* (*i*=0..log t) there is a buffer of 2^i buckets
- Each bucket contains O(log n) items



logn size per bucket since **whp** no more than logn items are mapped to each bucket

The Hierarchical Solution: Accessing an Item

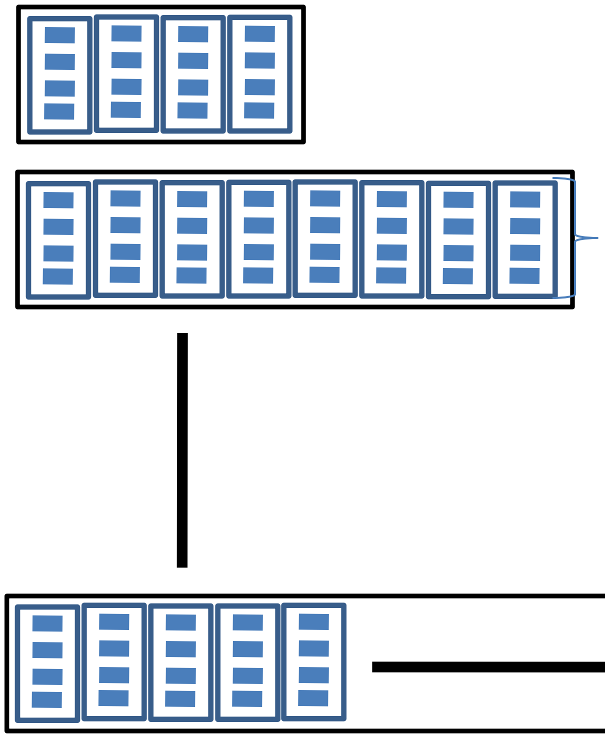


- Scan the entire first level for x
- Read a bucket from each other level, *i*=2,…,logn:
  - If x was not yet found, read and search bucket $H_i(x)$
  - If x was already found, search a random bucket

(If x found in more then one level, use top value of x)

The Hierarchical Solution: Writing Back an Item

- All items are written back to the first level
- If an item already exists there, rewrite it



The need for a reshuffle

- Each lookup ends with the updated item being written to the first level.

- At some point the first level becomes full
  - Then its contents are moved to the second level, which is twice as big.

The need for a reshuffle

- In general, level *i* stores at most 2^i items. (It has 2^i bins, each of size logn, storing the real items and padded with dummy items to size logn.)

- Every 2^i steps the (real) contents of level *i* are moved to level *i+1* and reshuffled with its contents.

- It always holds that levels *i* and *i+1* have together at most 2^{i+1} items.

Reshuffle

- The reshuffle process must
  - Empty level *i* and move its contents to level *i+1*
  - If an item with the same index *v* appears in both levels, its newest version (from level *i*) is kept and the other version is erased.
  - After the reshuffling, level *i+1* must be reordered using fresh random hash functions.

Implementing the Reshuffle

- Just use oblivious sorting:
  - Sort the contents of both levels based on their ids. A total of $(2^i+2^{i+1})\log n$ items.
  - $\Rightarrow$ Two copies of same item are now adjacent. Scan data and replace older copies with dummies.
  - Use a new hash function $H_{i+1}()$. Scan the items and attach $H_{i+1}(x)$ to each non-dummy item x.
  - Sort the contents. Whp at most logn items are assigned to each bucket.
  - Scan and adjust the number of dummies.

Hierarchical ORAM

- After the reshuffle level *i* is empty, and level *i+1* has at most 2^{i+1} items.
- A reshuffle of level *i* takes $O(2^i \log^2(2^i \log n)) = O(2^i \log^2 n)$ time.
- After n operations, the overhead of reshuffles is $O(n\log^2 n + 2\cdot(n/2)\log^2 n + 4\cdot(n/4)\log^2 n + \ldots) = O(n\log^3 n)$.
- Amortized cost of a lookup is therefore $O(\log^3 n)$

Hierarchical ORAM: Security

- Server's view is easy to simulate

- Accessing an element includes
  - A scan of the first level
  - Reading a random bucket in each level
  - Storing an item in the first level

Hierarchical ORAM: Security

- Level *i* is reshuffled every 2^i data accesses
- A reshuffle includes
  - Moving data
  - Oblivious sorts
  - Linear scans
- All operations are easy to simulate. The simulation breaks only if in some level more than logn items are mapped to the same bin.

Hierarchical ORAM: Discussion

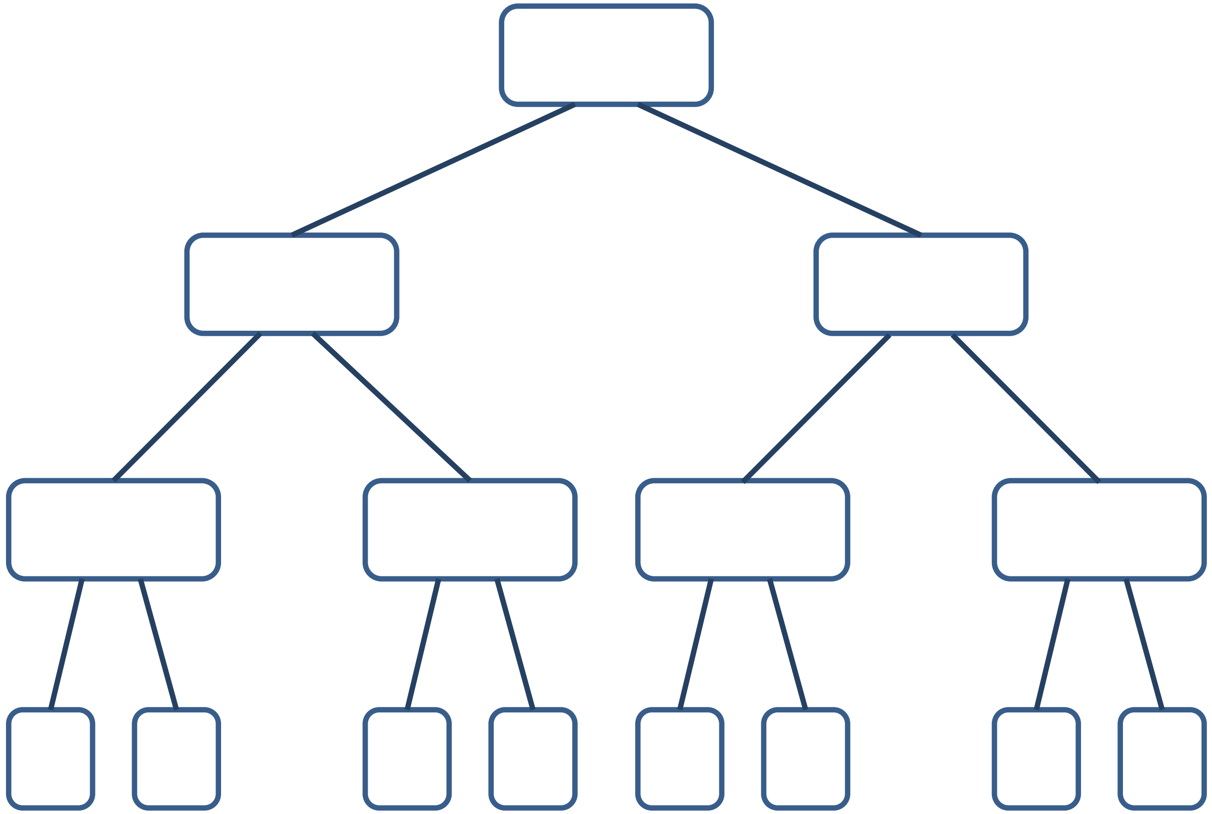
- Server storage is O(nlogn)
- The constants are quite high
- Amortized overhead of O(log^{3}n) hides a worst case time of O(nlog^{2}n) for a single operation.
- Replacing the Batcher sorting network with AKS removes a factor of logn from the asymptotic overhead, at the cost of a ridiculous constant.
- Other protocol variants exist.

Tree Based ORAM

Tree based ORAM

- A series of results that are very competitive and very simple to implement, in software and in hardware
  - *Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost*. E. Shi, T.-H. Chan, E. Stefanov, M. Li. Asiacrypt 2011.
  - *Path ORAM: An Extremely Simple Oblivious RAM Protocol*. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, S. Devadas. ACM CCS 2013.
- We will only describe the simplest scheme.

Server Storage



A full binary tree with logn levels and n leaves.

Each node contains a bucket of logn data items.

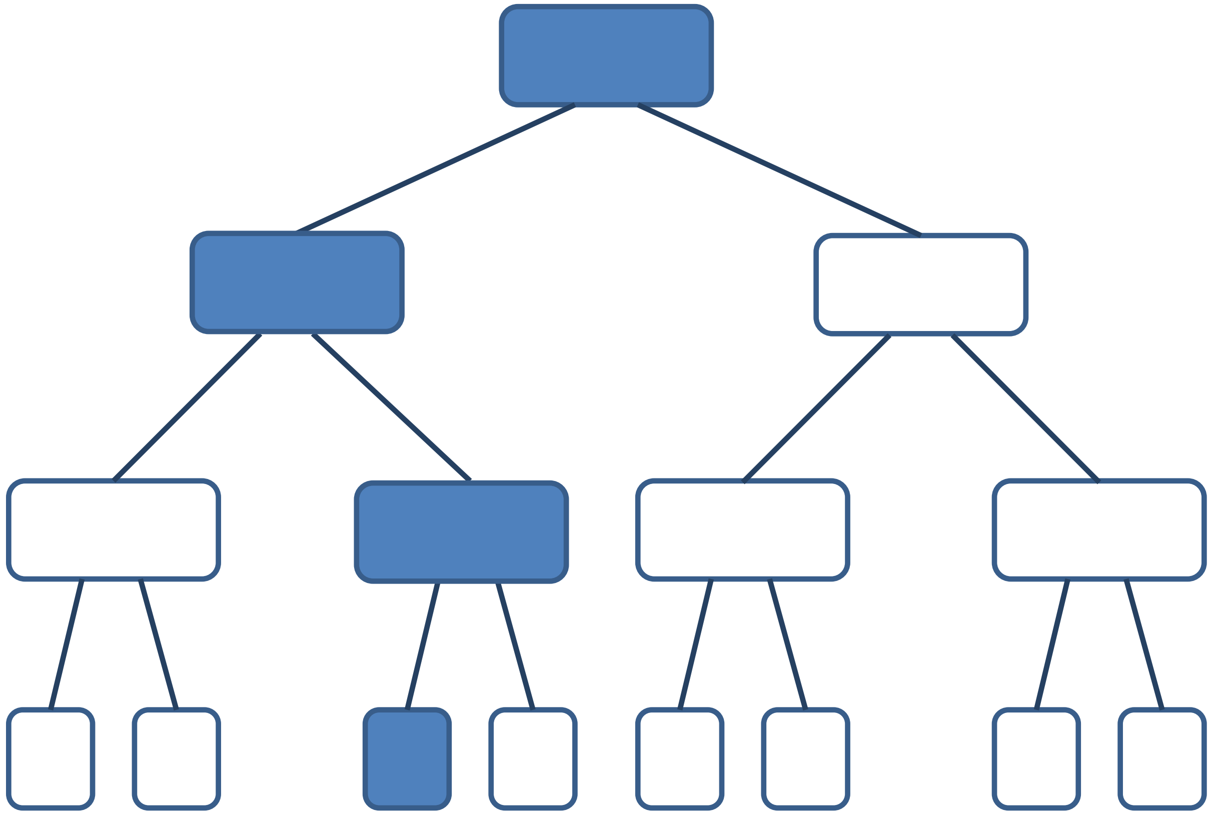
Client Storage

item	leaf
0	3
1	2
2	5
3	7
…	…
7	2

For now, assume that the client stores a position map, randomly mapping data items to leaves.

O(n) storage, but each item is only logn bits long.

Storing Items

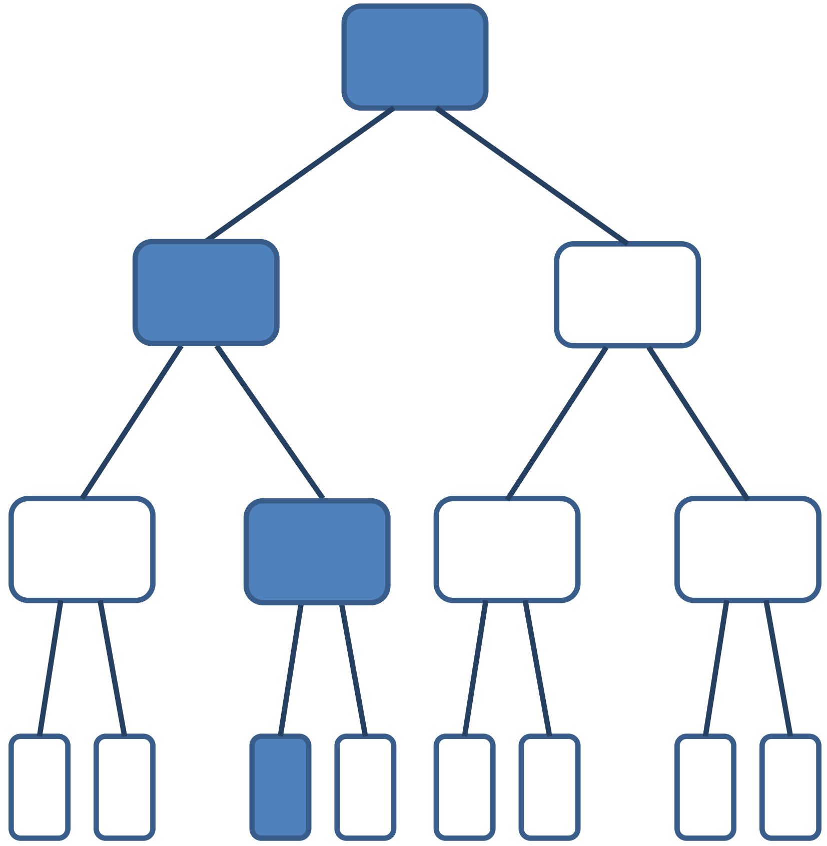


An item is always stored somewhere on the path from the root to its leaf.

item	leaf
0	3

Accessing an Item

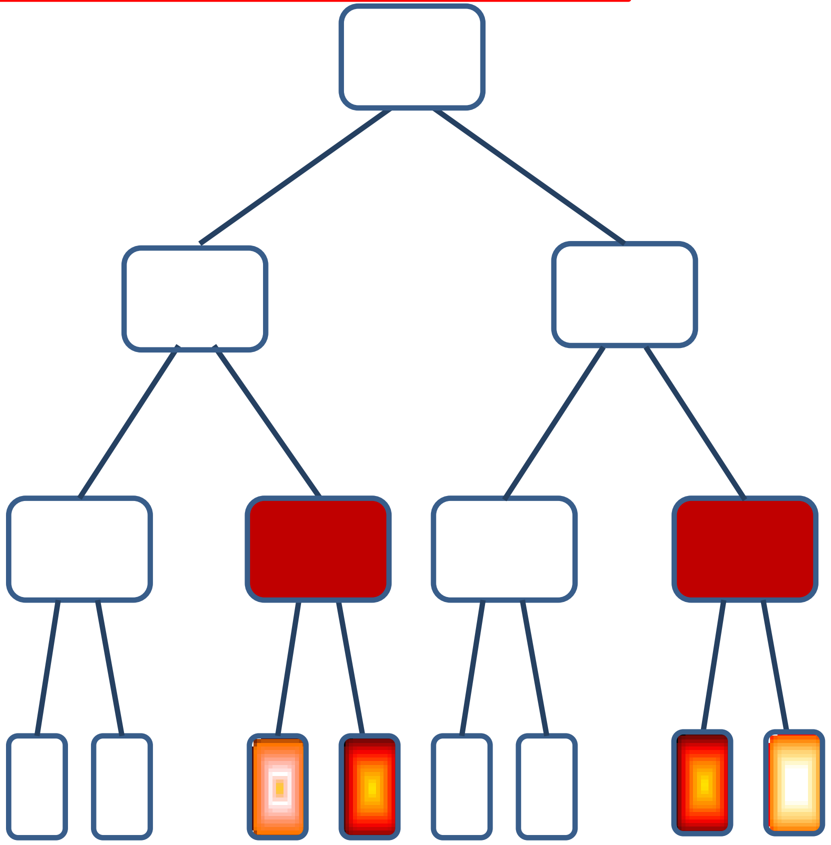
Note that these operations are oblivious



1. Read path (leaf) from position map.
2. Traverse path from root to leaf. Look for the item in each bucket along the path. Remove when found.
3. Assign a new random leaf to the data item.
4. Update position map.
5. Write updated item to the root.

Evict to Prevent Overflows

These operations are oblivious, too.



In each level choose two nodes at random

For each node

- Pop an item (if bucket is non-empty)
- Move item downwards to next node on its path
- Do a dummy write to other descendant of the node

Security

- All operations of the client are either deterministic or uniformly random
- All works well as long as no bucket overflows…
  - The evictions ensure this. The analysis uses Markov chains:
  - A buffer in level *i* receives an item with probability $(2/2^{i-1})\cdot(1/2)$
  - It evicts an item with probability $2/2^i$

Using Recursion (I)

- When the client looks for an item in a node, it can either
  - Read all O(logn) items in the bucket
  - Or, use ORAM recursively to check if the item it searches for is in the bucket

Using Recursion (II)

- In the basic scheme the client stores a position map of n·logn bits.
  - The client can store the position map on the server.
  - Its size is smaller than that of the original data by a factor of (data block length) / logn.
  - The client can access the position map using a recursive call to ORAM.
  - And so on…

Overhead

- Basic scheme
  - Server storage is $O(n \cdot \log n)$ data items
  - Client stores n indexes ($n \cdot \log n$ bits)
  - Each access costs $O(\log^2 n)$ r/w operations
- Using ORAM to read from internal nodes
  - Using, e.g., $n^{0.5}$-ORAM reduces cost to $O(\log^{1.5} n)$
- Storing position ORAM at server
  - Client storage reduced to $O(1)$
  - Overhead increases to $O(\log^{2.5} n)$

Followup Work

- Multiple results tweaking the construction
- Different variants
  - For small or large client storage (which can store O(logn) data items)
  - For small or large data items (blocks)
- Path ORAM achieves O(logn) overhead, with O(logn) client storage and *large* data items
  - Implemented even in hardware

Path ORAM

- Similar to the tree-based ORAM we described
- Eviction strategy is greedy:
  - The client maintains a stash of some data items
  - After searching for an item in path P, relocate each data item in P, as well as each item in the stash, as deep as possible along the path.
  - It was shown that this scheme works well even with buckets of size 4

Secure Computation based on ORAM

(Recall, a circuit implementing indirect memory access is inefficient. RAM machines are much better at this.)

Secure Computation based on ORAM [LO]

- Suppose two parties wish to securely compute a RAM program. The program
  - Has a state (shared by the parties)
  - Has a state machine (can be securely implemented by a circuit)
  - Needs to read/write a RAM

Secure Computation based on ORAM [LO]

- Read/write a RAM
  - Store RAM encrypted in P_1. P_2 knows the key.
  - The program accesses the RAM using ORAM.
  - The program state, shared by the parties, defines which RAM location to access. Therefore, the address to read/write is shared between P_1, P_2.
  - The ORAM “client” is now shared between the two parties.

Secure Computation based on ORAM [LO]

- Read/write a RAM
  - The operations of the ORAM "client" (data access, reshuffle, eviction) are implemented using secure computation.

Conclusions

- ORAM is a remarkable achievement and a great tool for many applications
- A huge amount of new results in recent years
  - At least 14 eprint manuscripts in 2014 alone
- Current performance is pretty impressive