Searchable Encryption Using ORAM

Benny Pinkas

Center for Research in Applied
Cryptography and Cyber Security

Desiderata for Searchable Encryption

- **Security**
  - No leakage about the query or the results
- **Functionality**
  - Variety of queries that are supported
- **Performance**

Using Secure Multi-Party Computation

- **MPC can be used for securely computing any function [Yao,GMW]**
- **In particular, the following function**
  - Client's input is a key, and an encrypted query
  - Server's input is a database encrypted with client's key
  - Client learns an output which is the result of the query
- **There are known techniques and libraries for implementing MPC**

Using Secure Multi-Party Computation

- **Pros:**
  - Fully secure, no leakage (except for upper bounds on the sizes of the inputs and output)
  - Full functionality
  - Can be made non-interactive using FHE
- **Cons:**
  - Performance: the “cryptographic overhead” is linear in the size of the database, which could be huge
  - Most efficient techniques (e.g., Yao) require using a fresh construction (circuit) for each query

Specific Constructions for Search on Encrypted Data

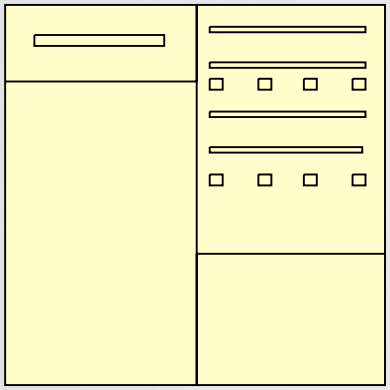
- **Deterministic encryption, order-preserving encryption, structured-encryption**
- **Pros:**
  - Very efficient
- **Cons:**
  - Leak some information
  - Partial functionality (targeted for answering specific types of queries)

Using Oblivious RAM for Search on Encrypted Data

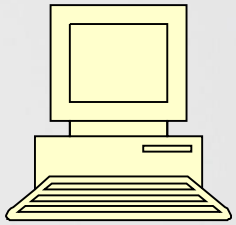
- **Security:** Leak less information than the specific constructions
- **Performance:** More efficient than MPC (polylog rather than linear overhead), but less efficient than specific constructions. Logarithmic # of rounds.
- **Functionality:** Less than MPC, more than specific constructions.

Oblivious RAM – the setting

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



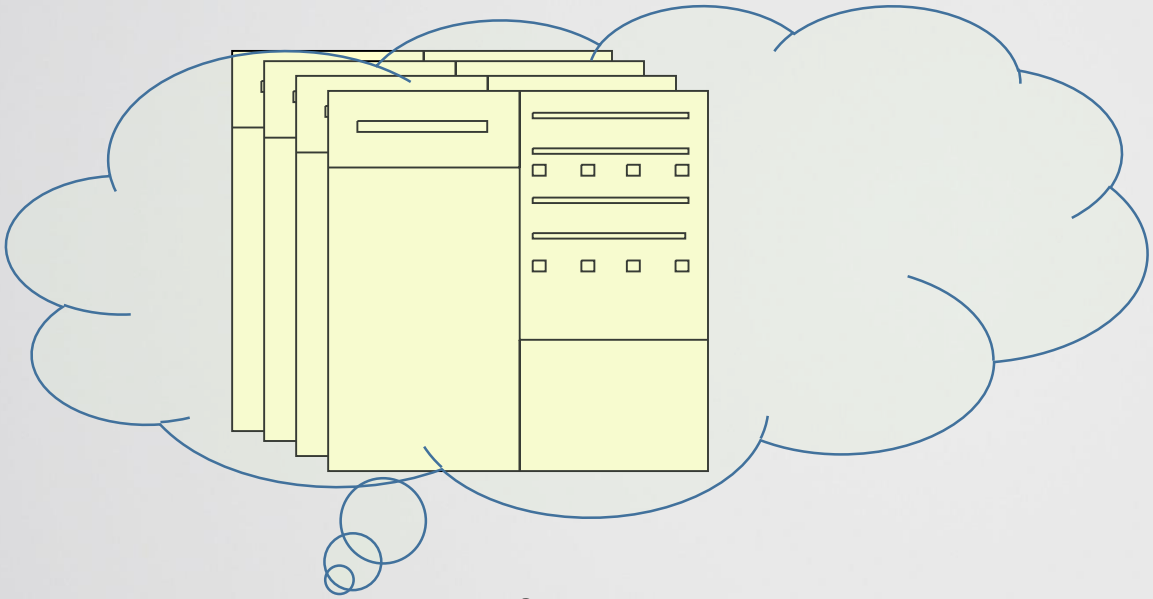
server



client

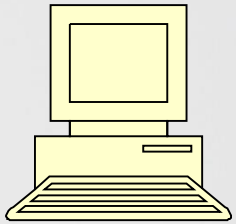
Oblivious RAM – the setting

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



Server farm

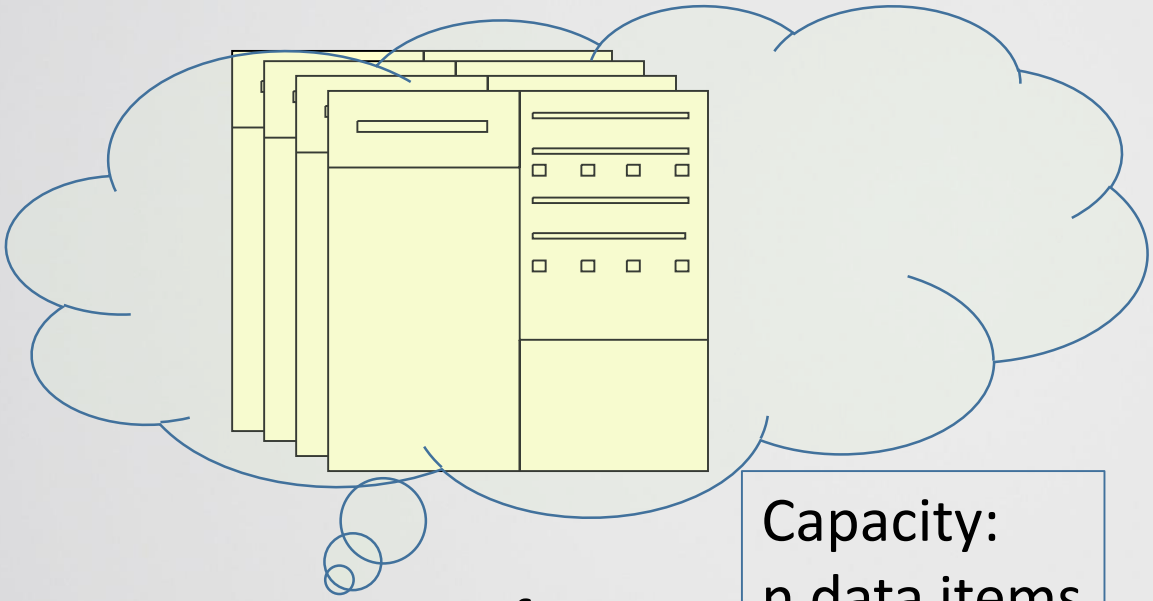
Cloud storage



Client

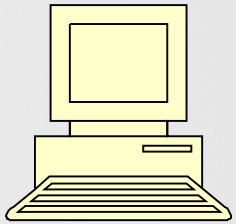
Oblivious RAM – the setting

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



Server farm
Cloud storage

Capacity:
n data items

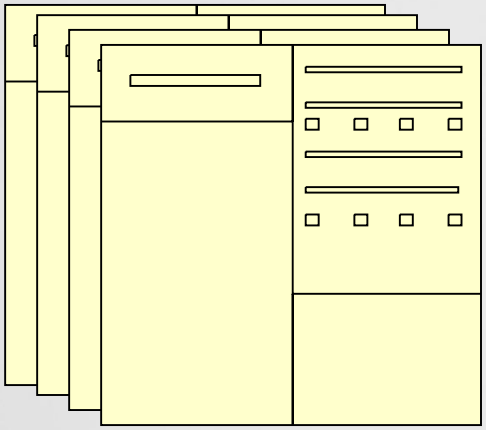


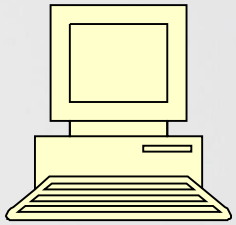
Capacity:
O(1) data items
log(n) bit *counter*

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 and MAC data to hide contents and prevent changes
  - **But the client also desires to hide access pattern to data**

Oblivious RAM – the setting

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

- Client can store data with the server
  - Can encrypt and MAC data to hide contents and prevent changes
  - **But the client also desires to hide access pattern to data**

Oblivious RAM - definition

- Client
  - Stores n data items, of equal size, of the form $(index_i, data\ block_i)$. $\forall i,j\ \ index_i \neq index_j$
  - 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")
- Preventing cache attacks (Osvik-Shamir-Tromer)
- Secure computation
- Search on encrypted data

Trivial ORAM 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**

ORAM - History

- Initial constructions by Goldreich and Ostrovsky (1987-1996).

- A very hot research topic in recent years
  - 807 Google Scholar articles containing “oblivious RAM”
  - 695 such articles since 2010

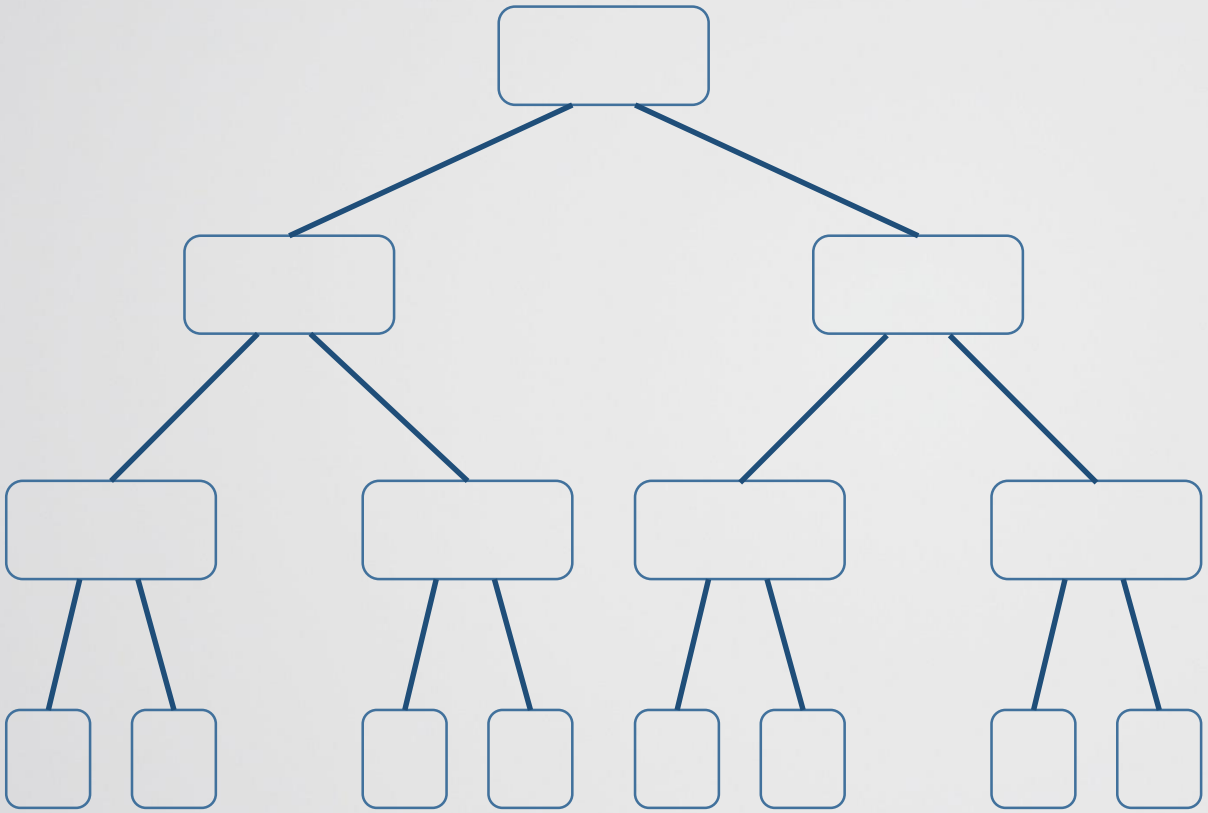
Tree Based ORAM Constructions

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

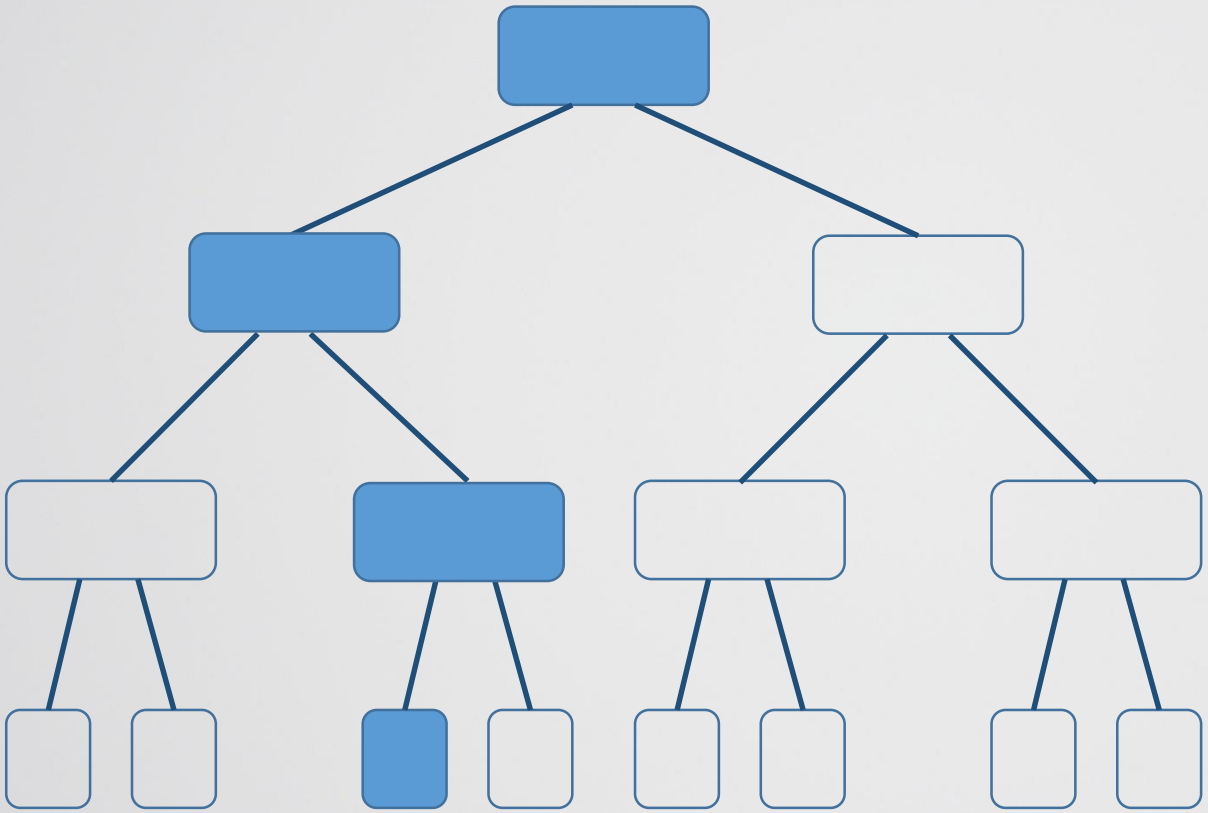
Client Storage

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

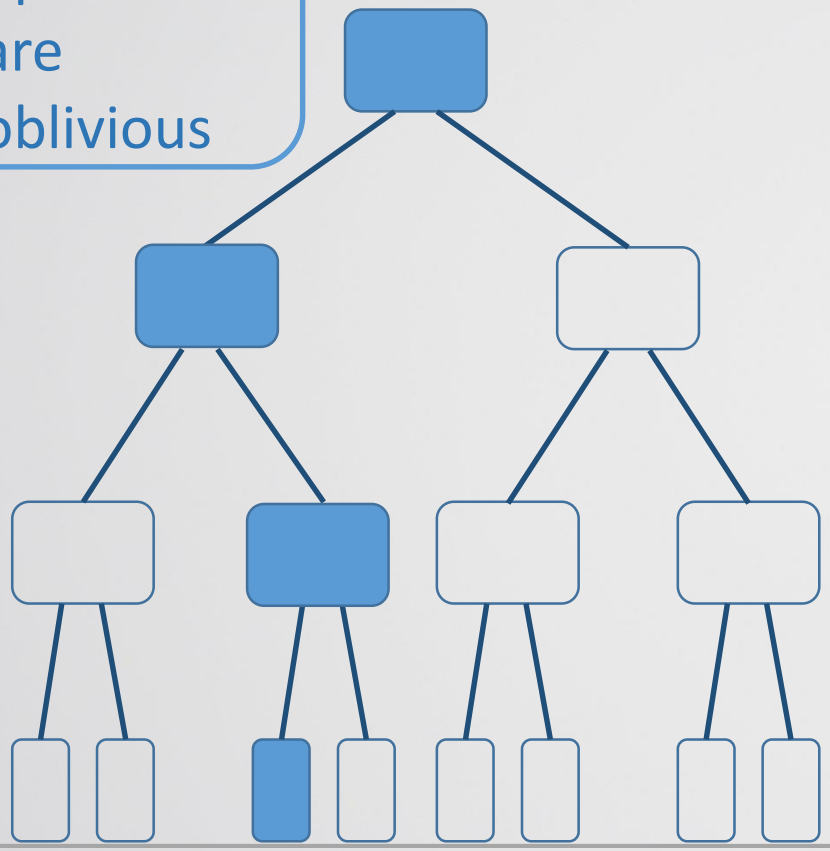


An item is always stored somewhere on the path from the root to its leaf

item	leaf
0	3

Accessing an Item

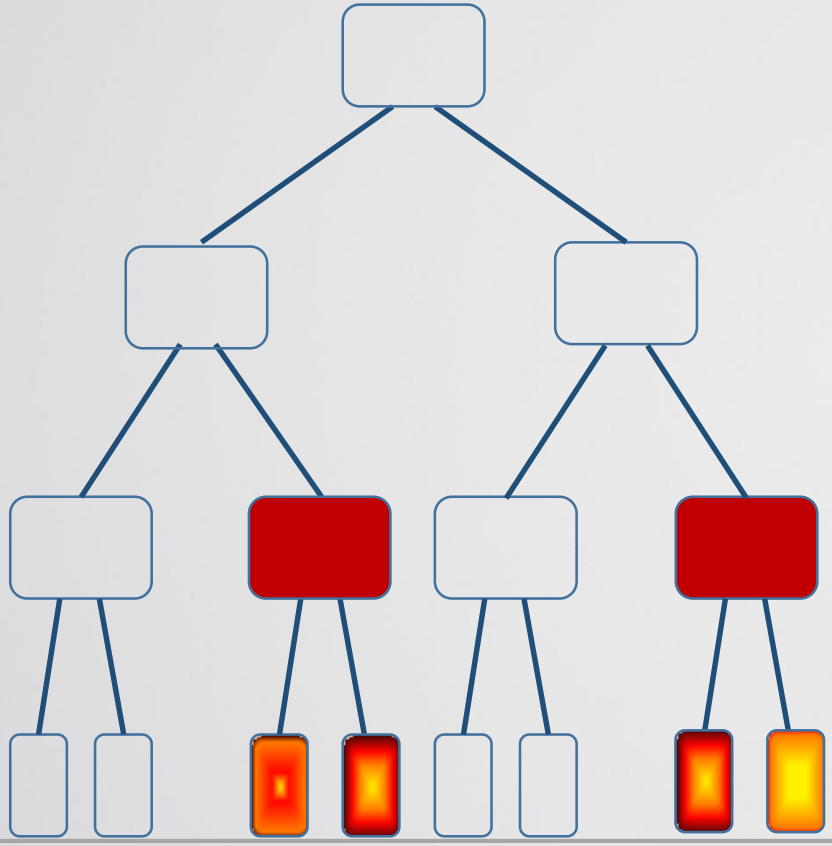
These operations are oblivious



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 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

Encrypted Search using ORAM

- **The (simplest) setting**
  - The client has *n documents* $X_1,\ldots,X_n$
  - There are *m* keywords $k_1,\ldots,k_m$
  - Each document is associated with a subset of the keywords
  - The data is stored encrypted at the server. The client has the encryption key.
  - The client wishes to retrieve all items associated with a specific key word *k*

Encrypted Search using ORAM

- Recall, in ORAM the client stores *n* data items, of equal size, of the form $(index_i, data\ block_i)$. $\forall i,j\ \ index_i \neq index_j$
- In our setting the client needs to search for an item using any of (the multiple) keywords associated with it

- An easy solution is to store tuples of the form $(k_i, a\ document\ associated\ with\ k_i)$, but this requires storing each document multiple times.

Encrypted Search using ORAM

- A solution: Use two ORAMs
  - The first ORAM stores a data structure that enables to compute; given a keyword, the documents associated with it.

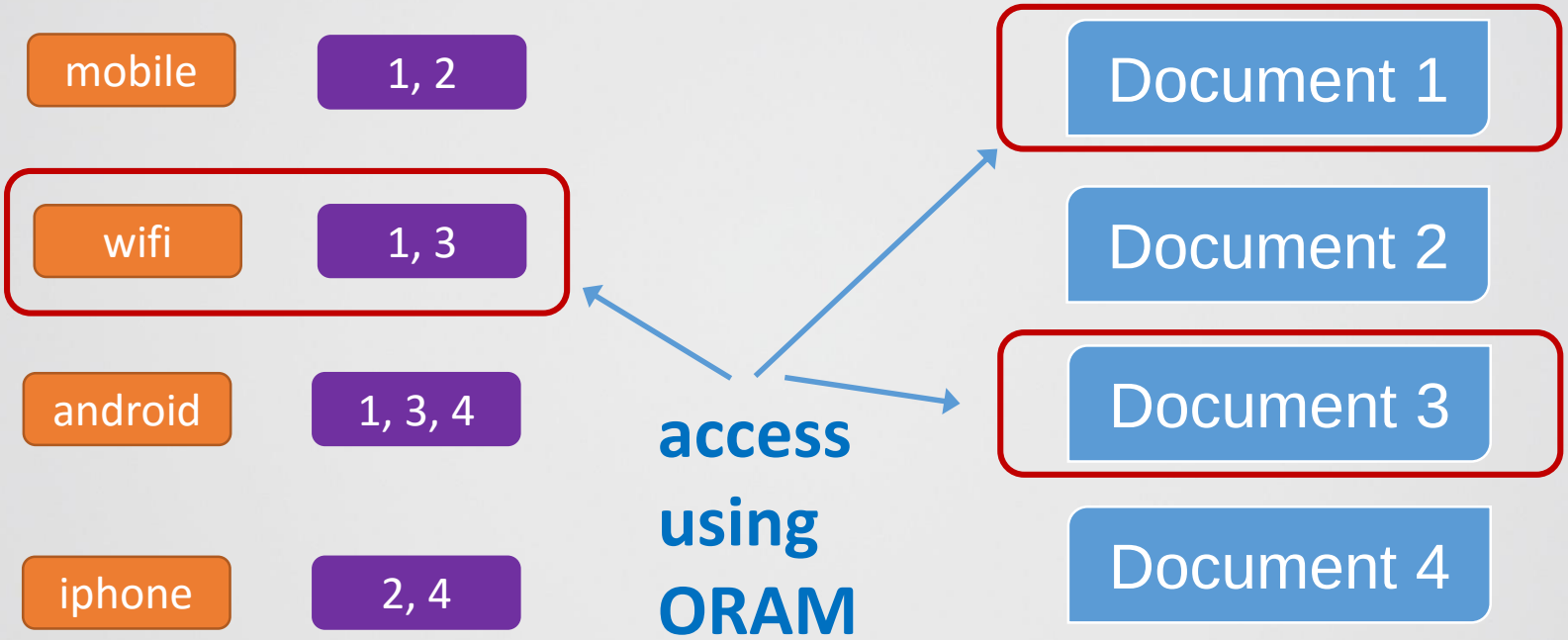
    (E.g., an inverted index)
  - The second ORAM stores the documents themselves.

Encrypted Search using ORAM

Encrypted Search using ORAM

Encrypted Search using ORAM

Searching for “wifi”



Generality

- The inverted index data structure can be replaced by any data structure supporting more complex queries (e.g., B-trees)

Security

- Client uses ORAM to read data stored at the server
  - Therefore the server cannot identify which item (keyword or document) is read…
  - The server cannot also identify if the same item is read twice…
- Or so it seems…

Efficiency Problems

- ORAM increases the communication and work overhead by a factor of at least log(n)
- Each ORAM access requires log(n) communication rounds (round-trip latency is often the bottleneck)
- Each ORAM read is for a block, which is wasteful since a block is often much larger than keyword data or than documents
  - Moreover, efficient ORAM schemes use larger blocks

Efficiency Problems

- ORAM increases the communication and work overhead by a factor of at least log(n)

× log(n)

- Each ORAM access requires log(n) communication rounds (round-trip latency is often the bottleneck)

× log(n) rounds

- Each ORAM read is for a block, which is wasteful since a block is often much larger than keyword data or than documents
  - Moreover, efficient ORAM schemes use larger blocks

× |block| / |data item|

Efficiency Problems

- Documents often have a Zipf distribution (power law)
- Simulations show that searching for any of the 5000 most frequent words in the Enron email database (of about 517,000 emails and 629,000 words) requires more communication than sending the database to the client.(1)

- Similar results hold for searching Wikipedia

(1) Muhammad Naveed, "The Fallacy of Composition of Oblivious RAM and Searchable Encryption, 2015.

Security Problems

- ORAM hides the access pattern to data
- But ORAM does not hide the **size** of the documents that are read
- ORAM-based encrypted search reveals
  - The size of the documents containing the query word
  - The number of documents containing the query word
  - The size of the inverted index and of the document set
  - And since documents and keywords have a Zipf law distribution, this helps identify them.

Lower Bound

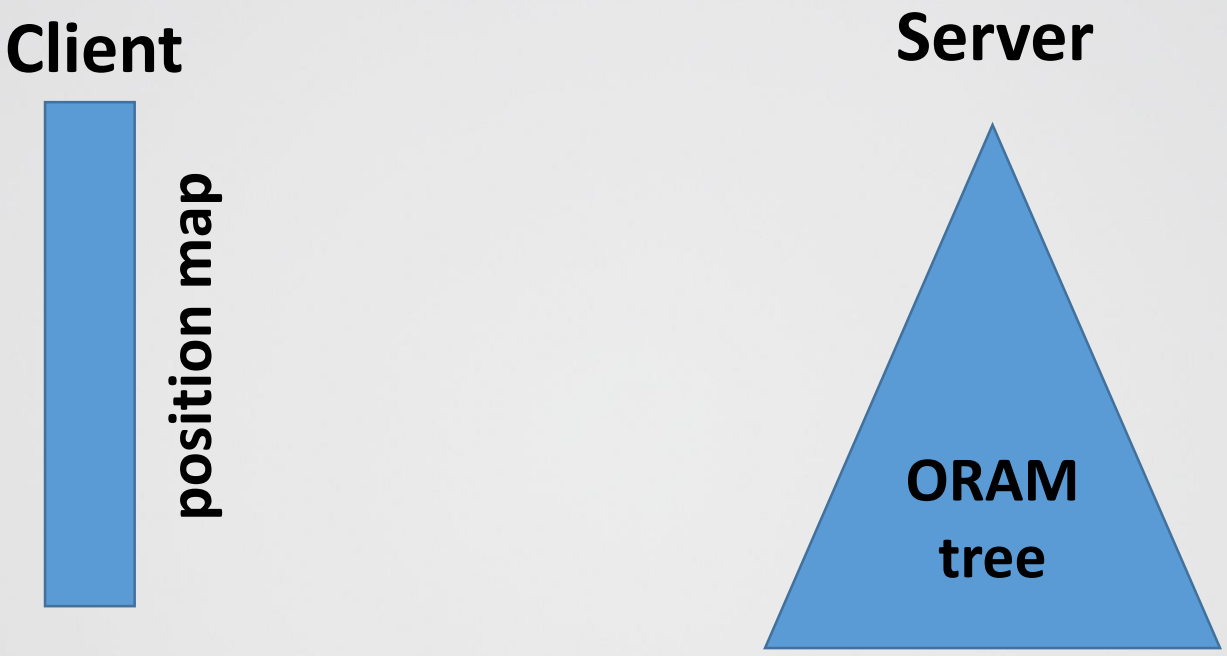
- Suppose we need an ORAM SSE that hides the total size of the documents containing the query word
  - Let Q1 be the query with the largest answer
  - Then for any other query, the answer must be as long as the answer for Q1

  - In the extreme, suppose Q1 is included in all documents.
  - Then for each query, the answer must be as big as the entire data set. (The server could just as well send all documents to the receiver).

ORAM SSE with only 2 rounds

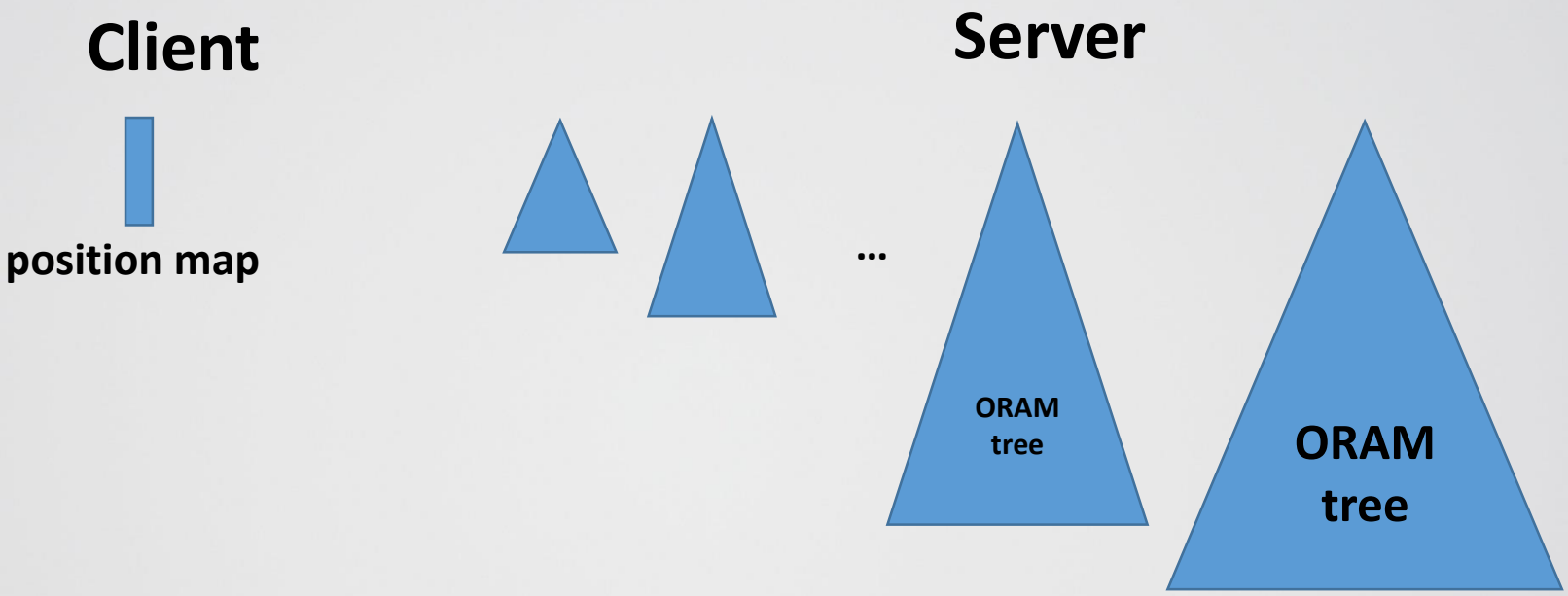
- Garg, Mohassel, Papamantou, “TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption”, 2015.

TWORAM: Recall basic ORAM



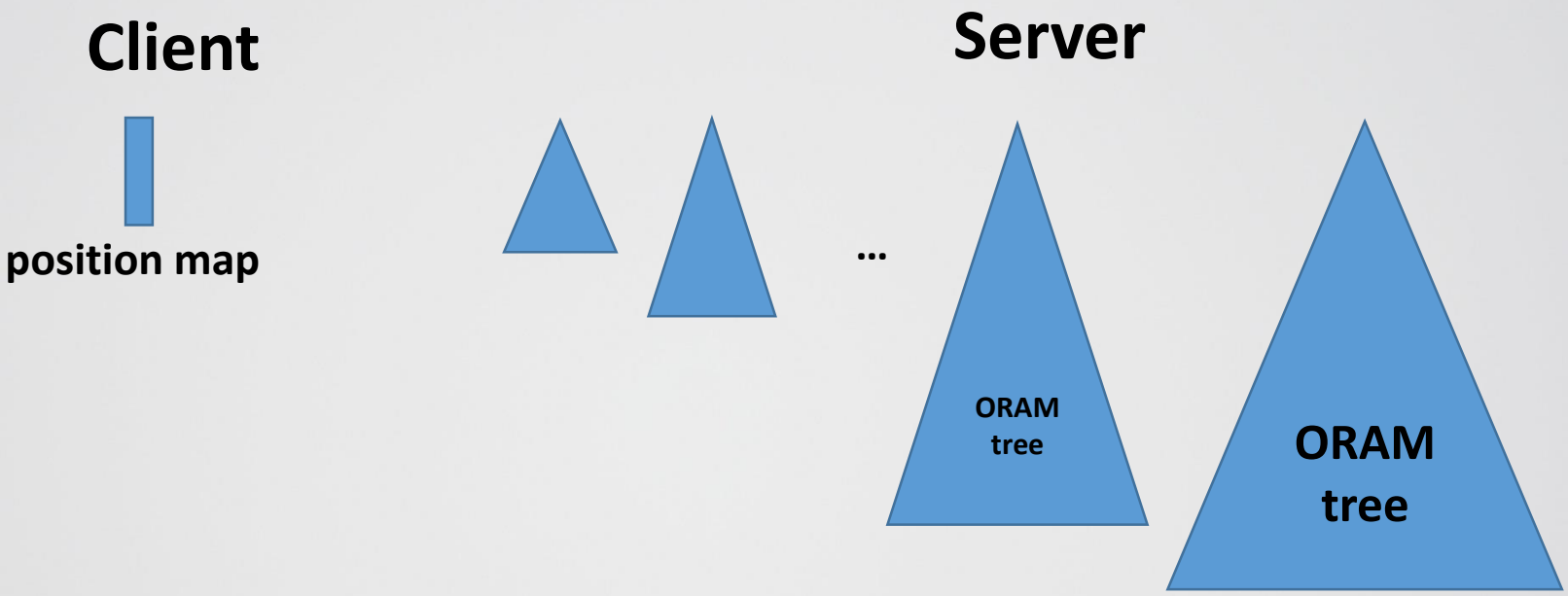
- In the basic tree ORAM scheme, the client stores n pointers to leaves

TWORAM: Recursive ORAM



In the recursive scheme, the server stores L = logn ORAM trees, storing smaller and smaller position maps

TWORAM: Recursive ORAM

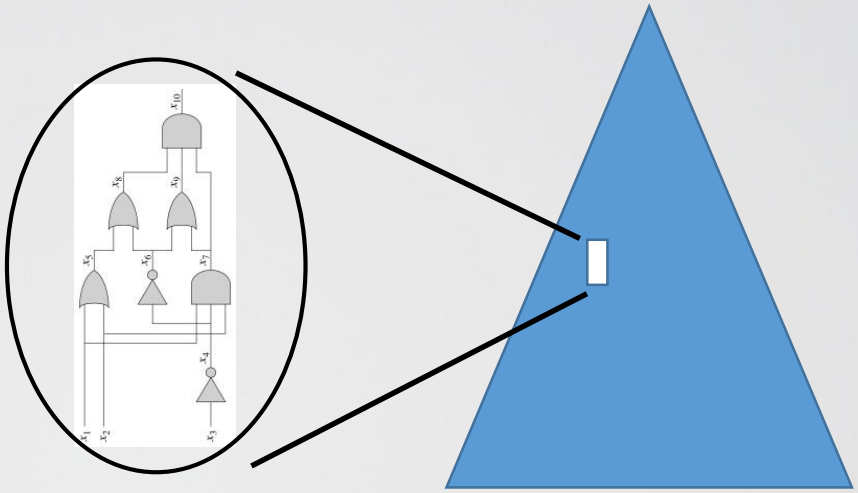


The client adaptively accesses these trees

=> log N communication rounds

TWORAM: The New Construction

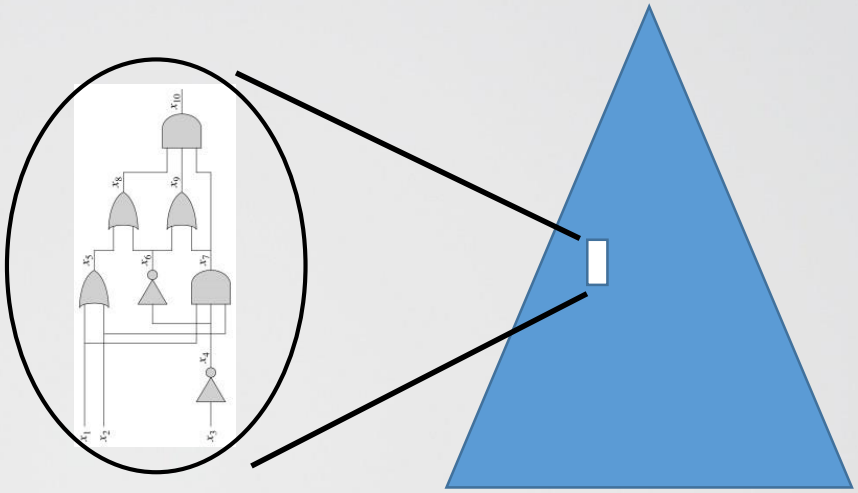
Replace each bucket in each tree with a **garbled circuit**



The circuit:

- Receives as input an item x that is looked for in the tree
- Has all values in this bucket hardcoded in the circuit
- If x is found in the bucket, outputs the path leading to x in the next tree

TWORAM: The New Construction

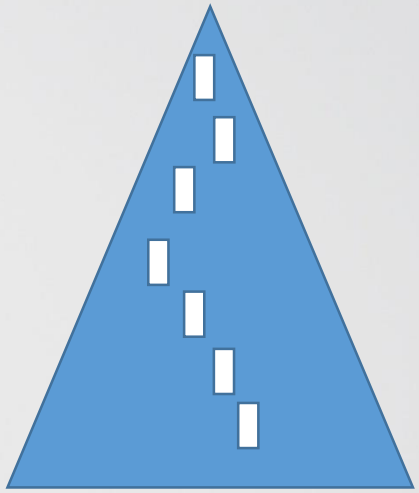


The server can evaluate the garbled circuit itself (using Yao's MPC protocol).

The server learns the output of the circuit, and uses it as its input to the next tree.

$\Rightarrow$ The server can search all trees without any interaction

TWORAM: Technical issues



Problem: must hide the exact bucket in which item was found

Solution: The output of bucket j is garbled, and is input to bucket j+1. Only the leaf bucket outputs the location of X in the next tree.

Problem: Once a garbled circuit is used it cannot be used again

Solution: At the end of the search the client sends new garbled circuits, replacing the circuits that were used in the search.

Search using TWORAM (naïve)

- Store the reverse index and the database in two separate TWORAMs

- Search:
  - Search the reverse index. Find out the locations of (say, S) relevant documents. *(O(1) rounds)*
  - Search the database for each of these documents *(O(S) rounds)* ☹

Better Search using TWORAM

- Assume database contains a total of N documents of the form $(k_i, doc_{i,j})$
  - Each keyword can be associated with many documents $(k_i, doc_{i,1})$, $(k_i, doc_{i,2})$, …
  - But total number of pairs is N

- The path associated with $(k_i, doc_{i,j})$ is a pseudo-random function of
  - The keyword k_i
  - The index j of $doc_{i,j}$
  - The number of times k_i was searched for

Better Search using TWORAM

Storing documents

- Store the pairs (k_i,$doc_{i,j}$) in a single-level path-ORAM
- There is no need for a position map since the position is a function of (k_i, j, #times k_i searched)

Storing the index

- Store in a TWORAM the tuples (k_i, #associated docs, #times k_i searched)

Better Search using TWORAM

Search

- Look for k_i in the TWORAM
- Retrieve $(k_i$, #associated docs, #times k_i searched$)$

- Compute all positions of $(k_i, doc_{i,j})$ using this information
- Look *in parallel* for all these documents in path-ORAM

O(1) rounds !

References

- Seny Kamara, How to Search on Encrypted Data: Oblivious RAMs (Part 4).
- Muhammad Naveed, “The Fallacy of Composition of Oblivious RAM and Searchable Encryption”, 2015.
- Garg, Mohassel, Papamantou, “TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption”, 2015.