Oblivious Algorithms

Brice Minaud

dvail: brice.minaud@ens.fr

MPRI, 2022
Course outline

“Techniques in Cryptography and Cryptanalysis”: will cover (a choice of) important areas of cryptography.

- Lattices ✔ done
- Zero-knowledge proofs ✔ done
- Oblivious algorithms now
Outsourcing storage

With SNARKs: could outsource computation.

Now: outsource storage. Benefits: availability, space, resilience...

The host server is honest-but-curious.
Outsourcing storage

**Setting:**

Client stores $n$ data items.

“Key-value store”: each item is pair (address, data).

Client makes read/write operations:

- $\text{read}(a)$: read data at address $a$; or
- $\text{write}(a, d)$: write data $d$ at address $a$.

Server sees accessed addresses, and data read/written.
Encryption

First, want to encrypt all data.

Before writing any data, encrypt it with semantically secure (IND-CCA) symmetric encryption.

→ The server learns nothing about the content of any data.

Including whether two data items are equal!
Symmetric Encryption.

Message space \( M \), ciphertext space \( C \), key space \( K \).

**Setup:** Pick key \( K \leftarrow \$ K \).

**Encryption:** encryption of \( M \in M \) is \( C = \text{Enc}_K(M) \in C \).

**Decryption:** decryption of \( C \) is \( M = \text{Dec}_K(C) \).

**Correctness:** for all \( M \in M \),

\[
\text{Dec}_K(\text{Enc}_K(M)) = M.
\]

**Security:** here, let us say IND-CCA.

**Caveats:**
- Deterministic scheme cannot be IND-CCA. Need randomness, or nonces.
- “Security” above only covers confidentiality, not integrity.
IND-CCA: indist. under Chosen-Ciphertext Attacks

Adversary

Repeat freely

Pick $M'_i$.  

Pick $C_i$.  

Pick $M_1, M_2$.  

Pick $M''_i, C'_i$...

with $C'_i \neq C^*$  

Compute $b'$.  

Challenger

Pick $K \leftarrow \{0,1\}^k$.  

Pick $b \leftarrow \$ \{0,1\}$  

$C^* = \text{Enc}_K(M_b)$  

$\text{Dec}_K(C_i)$  

$\text{Enc}_K(M'_i)$  

$M'_i$  

$M_1, M_2$  

$C_i$
**Conclusion:** hiding the content of data is easy with encryption.

But the server also sees the address accessed by the client: “access pattern”.

This reveals a lot of information about the client’s activities.
Simple example

Client = hospital, storing patient files.

Client often searches for all patients with age between $x$ and $y$.

What the server sees:

1. Client accesses files $a$, $b$, $c$.
2. Later, client accesses files $b$, $c$, $d$.

This is the only possible configuration (up to symmetry)!

$\rightarrow$ Server learns that age of files $b$, $c$ is between $a$ and $d$. 
Simple example

1. Client accesses files a, b, c.
2. Client accesses files b, c, d.
3. Client accesses files c, d.

Then the only possible order of ages is a, b, c, d (or d, c, b, a)!

**In general**: server learns order of records very quickly.

(If ages are in [0, N] and queries are uniform, the order is fully revealed after \(O(N \log N)\) queries.)

If every value in [0, N] appears, position in order reveals value → encryption was useless!
Approximate order learning

\(\varepsilon\text{-Approximate order reconstruction.}\)

\textbf{Roughly}: we learn the order between two records as soon as their values are \(\geq \varepsilon N\) apart. (\(\varepsilon = 1/N\) is full reconstruction)

Assuming uniform queries, this only requires \(O(\varepsilon^{-1} \log \varepsilon^{-1})\) queries.

\(\implies\) The server learns the order of all records within 5\% of \(N\) within a constant number of queries!
Further motivation

Beside storage outsourcing: many situations where memory accesses of a sensitive algorithm can be observed by adversary (fully, or partially).

- **Trusted Enclaves** (e.g. Intel SGX): Client = CPU, Server = RAM. (Original motivation.)
- **Cache attacks** (concurrent processes observing cache misses).
- **Searchable Encryption**.

See: Side-channel attacks.
Oblivious algorithm

**Oblivious algorithm:** an algorithm $A$ is **oblivious** iff for any two inputs $x$ and $y$, the memory accesses of $A$ on input $x$, and $A$ on input $y$, are **indistinguishable**.

**Next lecture:** will see ORAM, a technique to make any algorithm oblivious.
Oblivious RAM

Client stores $n$ data items.

“Key-value store”: each item is pair (address, data).

Client makes read/write operations:

- $\text{read}(a)$: read data at address $a$; or
- $\text{write}(a,d)$: write data $d$ at address $a$.

**Problem:** Client wants to do this without revealing which address is accessed.
Roadmap

1. Oblivious Sorting.

2. General ORAM.
Oblivious Sorting
Oblivious algorithm: an algorithm $A$ is oblivious iff for any two inputs $x$ and $y$, the memory accesses of $A$ on input $x$, and $A$ on input $y$, are indistinguishable.

Which of the following algorithms are oblivious? (assuming inputs are arrays of fixed size.)

1. Bubble Sort. ✔ yes
2. Quick Sort. ❌ no
3. Merge Sort. ❌ no
Basic operation: sorting two elements.

**Compare and swap**: on input \((x,y)\), if \(x < y\), output \((x,y)\), else output \((y,x)\).
Bubble Sort

\[ x \quad y \quad z \quad t \]

Diagram of bubble sort process.
**Sorting network**

**Comparator network:** A comparator network is an algorithm that consists in a sequence of compare-and-swaps ("comparators") between fixed inputs.

Can be represented in this form:

![Diagram](image)

**Sorting network:** A sorting network is a comparator network that correctly sorts its input (for all possible inputs).

*Remark:* testing whether a comparator network is a sorting network is co-NP-complete.
Size and depth

The **size** of a comparator network is its number of comparators.

The **depth** (or “critical path”) of a comparator network is the maximum number of comparators that an input value can go through. It is also the number of steps in a parallel computation of the network.

**Example:** comparator network of size 4 and depth 3.

→ Bubble sort is a sorting network of size $O(n^2)$ and depth $O(n)$. 
Can we do better?

**Proposition:** A sorting network must have size $\Omega(n \log n)$.

*Proof.* A network with $k$ comparators can permute its input sequence in at most $2^k$ different ways.

For a sorting network, we must have $2^k \geq n!$

By Stirling’s formula, this yields $k = \Omega(n \log n)$.

**Bitonic sort:** size $O(n \log^2 n)$.

Most efficient in practice.
The 0-1 principle

0-1 Principle: a comparator network (on \( n \) inputs) is a sorting network iff it correctly sorts all \( 2^n \) possible binary inputs.

Proof. Let \( f \) be a non-decreasing function: \( x \leq y \) implies \( f(x) \leq f(y) \).

Claim: If a comparator network has input \((x_1, \ldots, x_n)\) and outputs \((y_1, \ldots, y_n)\), then on input \((f(x_1), \ldots, f(x_n))\), it must output \((f(y_1), \ldots, f(y_n))\).

Proof of the claim: induction on comparators.

Now assume a comparator network is not a sorting network. Then there exist an input \((x_1, \ldots, x_n)\) and some indices \(i, j\), such that \(x_i < x_j\) but they are in the opposite order in the output.

Define \( f(x) = 0 \) if \( x \leq x_i \), \( 1 \) otherwise. We have \( f(x_i) = 0 \) and \( f(x_j) = 1 \), but their order is reversed by the network when inputting \((f(x_1), \ldots, f(x_n))\). Hence the network does not correctly sort all binary sequences.
Bitonic sequences

Bitonic sequence: A sequence of values is bitonic iff:

- It is increasing, then decreasing.
- Or it is a circular shift of the previous case.

Example: bitonic sequences of 0 and 1’s are those of the form $0^a1^b0^c$ and $1^a0^b1^c$. 
**Half-cleaner**

**Half-cleaner**: A half-cleaner is a comparator network for an even number of inputs $n$, composed of comparators

$$(1, n/2+1), (2, n/2+2), \ldots, (n/2, n)$$

Half-cleaner for $n = 8$:
**Half-cleaner**

**Key property of a half-cleaner:** if the input is bitonic, then both halves of the output are bitonic. Moreover, one of the two halves must be all 0’s or all 1’s. That half is called **clean**.
The bitonic sorter correctly sorts all bitonic inputs.
Batcher’s sort correctly sorts all binary inputs, hence all inputs.
A half-cleaner has size $H(n) = n/2$.

The bitonic sorter has size $S(n) = H(n) + 2S(n/2) = O(n \log n)$.

Batcher’s sort has size $B(n) = S(n) + 2B(n/2) = O(n \log^2 n)$.

The depth of Batcher’s sort is $O(\log^2 n)$: in a parallel computation model, only need $O(\log^2 n)$ steps.

→ Sorting algorithms used in GPUs.

**Ajtai, Komlós, Szemerédi (STOC ’83):** there exists a sorting network of size $O(n \log n)$.

Unfortunately, completely impractical.
Oblivious RAM
So far…

Traditional efficient sorting algorithms were not oblivious.

→ created new efficient oblivious sorting algorithm.

Can we do this generically?

Take any algorithm → create oblivious version, with low overhead.

This is what Oblivious RAM (ORAM) does.

Disclaimer: does not hide number of accesses.
Reminder: Oblivious RAM

Client wants to do queries \( q_1, q_2, \ldots, q_n \).

Each \( q_i \) is either:
- \text{read}(a): \text{read data block at address } a;
- \text{write}(a,d): \text{write data block } d \text{ at address } a.
**Reminder: Oblivious RAM**

**ORAM algorithm** $C$ (or ORAM “compiler”): transforms each query $q$ by the client into one or several read/write queries $C(q)$ to server.

**Correctness:** $C$’s response is the correct answer to query $q$.

**Obliviousness:** for any two sequences of queries $q = (q_1,\ldots,q_k)$ and $r = (r_1,\ldots,r_k)$ of the same length, $C(q) = (C(q_1),\ldots,C(q_k))$ and $C(r) = (C(r_1),\ldots,C(r_k))$ are indistinguishable.
**Trivial ORAM:** read and re-encrypt *every* item in server memory.

**Security:** trivial.

**Efficiency:** every client query costs $O(n)$ real accesses $\rightarrow$ overhead is $O(n)$.

A non-trivial ORAM must have:

- Client storage $o(n)$.
- Query overhead $o(n)$.
Some observations

Suppose client wants to do queries $q_1, q_2, \ldots, q_n$.
Each $q_i$ is to read or write a block of memory.

Assume the client does not store any memory block.

For each $q_i$, the ORAM has to do some access(es) to the server memory.

$q_1$ and $q_2$ must access at least 1 data block in common.
Some observations

$(q_1, q_2)$ and $(q_3, q_4)$ must access at least 2 data blocks in common.
Some observations

etc...

$q_1$ $q_2$ $q_3$ $q_4$ $q_5$ $q_6$ $q_7$ $q_8$ ... $q_n$
Some observations

For each memory access done by $q_j$, let's "assign" that access to the node $q_i \land q_j$, where $q_i$ is the last time the same address was accessed ($i < j$).

At least this many accesses are assigned to this node.
Some observations

For each memory access done by \( q_j \), let's "assign" that access to the node \( q_i \land q_j \), where \( q_i \) is the last time the same address was accessed (\( i < j \)).

→ Memory accesses in every node are now unique to that node.
Some observations

Wait... how many memory accesses are we doing?

(Say $n = 2^k$ for some $k$.)

For $n$ client queries, ORAM will need to do $\Omega(n \log n)$ accesses to the server.
A lower bound

Goldreich & Ostrovsky ’96 (again):
Secure ORAM must have overhead $\Omega(\log n)$.

G&O's proof under assumptions:
- Client memory $O(1)$.
- Statistically secure ORAM.
- “Balls and bins” model.

What we just saw: stronger proof by Larsen & Nielsen ’18.
(Computational security, less restrictive cell probe model, online).
Proof sketch (Goldreich-Ostrovsky proof)

Each item $i = \text{colored ball}$. At start:

Client: $c = O(1)$ balls  Server: $n$ balls + extra room

Suppose client wants to make queries for balls $b_1, \ldots, b_q$.
→ ORAM makes accesses $a_1, \ldots, a_{f(q)}$. (Includes Setup accesses.)

Each server access, ORAM can do $O(c)$ operations: exchange ball, put ball, take ball, nothing.

Statistical security → access sequence $(a_i)$ must be compatible with all $n^q$ possible query sequences $(b_i)$.

But only $O(c)^{f(q)}$ possible sequences of balls held by client, hence $O(c)^{f(q)}$ query sequences compatible with given access sequence.

\[ \Rightarrow \quad O(c)^{f(q)} \geq n^q \]

\[ \Rightarrow \quad f(q) = q \cdot \Omega(\log n) \]
**Roadmap**

**Query overhead**: how many queries to the server are made in \( C(q) \) for each client query \( q \), amortized (= on average).

Here, \( n = \text{max memory size} = \text{max number of items} \ (address, data) \).

<table>
<thead>
<tr>
<th>Family of constructions</th>
<th>Overhead</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Square-root ORAM</td>
<td>( \tilde{O}(n^{1/2}) )</td>
<td>Simple</td>
</tr>
<tr>
<td>2. Hierarchical ORAM</td>
<td>( O(\text{polylog } n) )</td>
<td>Best in theory</td>
</tr>
<tr>
<td>3. Tree ORAM</td>
<td>( O(\text{polylog } n) )</td>
<td>Best in practice</td>
</tr>
</tbody>
</table>

\( \text{polylog}(x) = \text{poly}(\log(x)) = O(\log^c(x)) \) for some constant \( c \).

**Other efficiency metrics**: client memory size, number of roundtrips in \( C(q) \), time complexity of \( C \)…
Square-root ORAM

Goldreich and Ostrovsky ’96.

Let $s < n$. (Later, will fix $s \approx n^{1/2}$.)

Want to store $n$ items. Create room for $n+2s$ items:

The **main memory** stores $n+s$ items:

- **Real items**: with addresses in $[1,n]$, real data.
- **Dummy items**: with addresses in $[n+1,n+s]$, random data.

For now, **stash** contains $s$ items with all-zero address and data.
1. Client chooses permutation $\pi$ over $[1, n+s]$.

**Item $i$ will be stored at location $\pi(i)$ in the main memory.**

2. Client encrypts everything, and sends to server.

Server view:

- $n+s$ encrypted items
- $s$ enc. items

*Remark*: we are assimilating client with ORAM algorithm.
Set $t = 1$. (number of dummy items read so far)

To access (= read/write) item $i$, client does:

1. Read the whole stash.

   *If item $i$ was not found in stash:*
   2. Read/rewrite location $\pi(i)$ in main memory.
   3. Add item $i$ to stash, rewrite whole stash to server.

   *If item $i$ was found in stash:*
   2. Read/rewrite location $\pi(n+t)$ in main memory. $t \leftarrow t+1$.
   3. Rewrite whole stash to server.
Refresh

Lookup can fail in two ways: stash is full, or run out of fresh dummy items ($t > s$).
Can only happen after $s$ iterations.

**Solution:** after $s$ iterations of *lookup*, perform *refresh*:
- Client chooses new permutation $\pi'$.
- Moves item $i$ to location $\pi'(i)$ in main memory.
- Empties stash.

→ equivalent to fresh setup with $\pi'$.
→ can do $s$ iterations again…

How do you move item $i$ to location $\pi'(i)$ obliviously?

**Oblivious sorting!**
Refresh via oblivious sort

Server memory after $s$ lookups...

- Main memory: $n$ real items (some outdated), $s$ dummies
- Stash: $n$ real items, empty items

\[ \text{Oblivious sort with } \pi^{-1} \]

- $n$ real items (some duplicates)
- Stash: $s$ dummies, empty items

\[ \text{Erase outdated duplicates} \]

- $n$ real items + some empty
- Stash: $s$ dummies, empty items

\[ \text{Oblivious sort with } \pi' \]

- $n+s$ items sorted with $\pi'$
- Main memory
- Stash: empty items
Security

Setup: server sees:

- main memory
  - $n+s$ encrypted items

- stash
  - $s$ enc. items

Lookup: server sees:

- main memory
  - Access to uniformly random fresh location

- stash
  - Full rewrite

Refresh: server sees:

1 oblivious sort, 1 linear scan, 1 oblivious sort.

Remark: computationally secure. Essentially statistically secure, except for encryption, and pseudo-random permutation $\pi$. 
Efficiency

Overhead.

Lookup costs $O(s)$.

Refresh costs $O(n \text{ polylog } n)$, happens every $s$ lookups.

Total overhead (amortized):

$$O( s + n/s \cdot \text{polylog}(n) )$$

Setting $s = n^{1/2} \log n$, and using Batcher sort:

$$O( n^{1/2} \log n )$$

Server memory: $O(n)$.

Client memory: $O(1)$.

Need encryption key + key for pseudo-random $\pi$ + few items during operations.

Remark: memory measured in number of items. Item size assumed to be $\Omega(n)$ bits, which is also $\Omega(\lambda)$ if $n \geq \lambda$. 

50
Hierarchical ORAM
Hierarchical ORAM

Goldreich and Ostrovsky ’96:
- Square-root ORAM, overhead $\tilde{O}(n^{1/2})$.
- Secure ORAM must have overhead $\Omega(\log n)$.

But also: hierarchical ORAM, overhead $O(\log^3 n)$.
→ Spawned whole construction family of ORAMs.

Interesting because:
- First ORAM with polylog overhead.
- Basis for the recent construction of optimal ORAM with overhead $O(\log n)$.

Hashing

Hash function $H: \{0,1\}^* \rightarrow [1,n]$.

Want to store $n$ items into $n$ buckets according to $H$.

Buckets of size $\log n$ suffice for negligible probability of overflow.

**Proof:** Probability that given bucket receives more than $k$ items is $\exp(-\Omega(k^2))$ by Chernoff bound. Union bound over all buckets:

$$n \cdot \exp(-C \cdot \log^2 n)) = n^{1 - C \cdot \log n} = \text{negl}(n).$$
Oblivious hashing

Want to do the assignment obliviously...

Suppose we have items + empty buckets all in server memory.

Assignment can be done obliviously in $n \log^2 n$ operations.

**Sketch:**
1. obliviously sort items according to $H$.
2. Put each item into own bucket.
3. Scan all buckets, pushing content of each bucket into next bucket if next bucket has same hash value.
4. Obliviously sort *buckets* to delete empty buckets.
Server memory arranged into log $n$ levels.
Each level $k$ is an (oblivious) hash table for $2^k$ items.

At start:
All items are in last level.
Other levels contain dummies.
To access item item $i$:

1. Access each level $k$ at location $H_k(i)$ until item is found.
2. Access remaining levels at uniformly random location.
3. Insert item at level 1. (Potentially with new value.)

*Remark:* whenever accessing level 1, entire level is read + rewritten.
To maintain invariant that level $k$ stores $\leq 2^k$ items:

Every $2^k$ lookups, the (non-dummy) items of level $k$ are shuffled into level $k+1$, using fresh hash function.

If an item appears twice, newest version (from earliest level) is kept.

**Invariant is preserved:**
Level $k$ receives at most $2^{k-1}$ items every $2^{k-1}$ lookups.
And empties its content every $2^k$ lookups.

*Remark:* last level is never full, because it can hold $n$ items, and there are no duplicate items in the same level.
Security

**Setup**: server sees \( \log n \) hash tables:

- size 2
- size 4
- size 8
- ...

**Lookup**: server sees:

- size 2
- size 4
- size 8
- ...

- full rewrite
- uniformly random reads
- + oblivious reshuffles at predetermined times.

**Key fact**: no item is ever read twice from the same level with the same hash function.
Efficiency

**Overhead.**

Level $k$ is reshuffled every $2^k$ lookups.

Each reshuffle costs: $O(2^k \log^2 n)$.

→ Amortized cost for level $k$: $O(\log^2 n)$.

→ Total amortized cost of reshuffles: $O(\log^3 n)$.

→ Total amortized overhead: $O(\log^3 n) + O(\log^2 n) = O(\log^3 n)$.

**Server memory:** $O(n \log n)$.

**Client memory:** $O(1)$.

Server memory can be reduced to $O(n)$ using *cuckoo hashing*.
Cuckoo hashing

“Bucket” hashing had total storage $O(n \log n)$, and lookup $O(\log n)$. Cuckoo hashing has storage $(2+\epsilon)n = O(n)$, and lookup $2 = O(1)$.

Initial design mainly motivated by real-time systems…

Idea:

Each item $i$ can go into one of two cells $H_1(i)$ or $H_2(i)$. 

$m=O(n)$ cells
The cuckoo graph

Picture graph with cells = nodes, item $i = \text{edge } H_1(i) - H_2(i)$. 
The cuckoo graph

Picture graph with cells = nodes, item \( i \) = edge \( H_1(i) - H_2(i) \). Orient edge towards where item is stored.

To insert item \( i \): try cell \( H_1(i) \). If occupied, move occupying item into its other possible cell. Repeat until unoccupied cell is reached.
The cuckoo graph

Picture graph with cells = nodes, item $i = \text{edge } H_1(i) - H_2(i)$. Orient edge towards where item is stored.

To insert item $i$: try cell $H_1(i)$. If occupied, move occupying item into its other possible cell. Repeat until unoccupied cell is reached.
Why does that work? (sketch)

**Theorem:** assignment is possible iff every connected component has at most one cycle.

Moreover, with \( n \) edges and \( m = (2+\varepsilon)n \) nodes…

- The previous fact holds with high probability.
- Expected size of a connected component is \( O(1) \).

\[ \rightarrow \text{Expected insertion time is } O(1)! \]

**Remark:** Probability of failure can be made negligible by adding a stash.
Tree ORAM
Hierarchical ORAM family leads to recent optimal construction. But huge constants. Never used in practice.

What is actually used:

Tree ORAM

Overhead: $O(\log^3 n)$.
Worst-case (no need to amortize).

In practice: easy to implement, efficient.

We will see Simple ORAM, member of the Tree ORAM family.
Server-side memory is a full binary tree with log\((n/\alpha)\) levels.

Each node contains log \(n\) blocks.

Each block contains \(\alpha = O(1)\) (possibly dummy) items.
Items are grouped into blocks of $\alpha$ items, item $i$ into block $b = \lfloor i/\alpha \rfloor$.

**At start:**
Each block $b$ is stored in a uniformly random leaf $\text{Pos}(b)$. “Position map” $\text{Pos}()$ is stored on the client.

**Invariant:** block $b$ will always be stored on the branch to $\text{Pos}(b)$.
To access item $i$ from block $b$:

1. Read every node along branch to $\text{Pos}(b)$. Remove $b$ when found.
2. Update $\text{Pos}(b)$ to new uniform leaf.
3. Insert $b$ at root. (Possibly with new value.)
Eviction

After every lookup

1. Pick branch to uniformly random leaf.
2. Push every block in the branch as far down as possible (preserving that block $b$ must remain on branch to $\text{Pos}(b)$).
Security

Setup: server sees full binary tree of height $\log(n/\alpha)$. Each node is encrypted, same size.

Lookup + eviction: server sees:

Full read/rewrite along 2 branches to uniformly random leaves.
Why does that work? (sketch)

Works as long as no node overflows.

**Setup**, no overflow: same argument as bucket hashing.

**Lookup + eviction**, no overflow (sketch):
Let $K$ be the number of blocks per node (we had $K = \log n$).
Pick arbitrary node $x$ at level $L$.
For $x$ to overflow, number of blocks whose Pos is below $x$ must be at least $K$.
$\rightarrow$ For one of the two children of $x$, number of blocks whose Pos is below that child $c$ must be at least $K/2$.
$\rightarrow$ This implies event [Pos of new block is below $c$] happens $K/2$ times, without event [eviction branch includes $c$] happening at all.
Both events have the same probability (namely $2^{-L}$).
Deduce overflow probability is $\leq 2^{-K/2}$. Negligible for $K = \omega(\log n)$.

*Remark:* we cheat a little by setting $K = \log n$. 
Efficiency of basic construction

**Overhead.**
Each lookup, read two branches, total $O(\log^2 n)$ items.

**Server memory:** $O(n \log n)$.

**Client memory:** $O(n/\alpha)$. (oops)
The position map

The client stores position Pos: $[1, n/\alpha] \rightarrow [1, n/\alpha]$, size $n/\alpha = \Theta(n)$. Still a large gain, if item size is much larger than $\log(n/\alpha)$ bits.

To reduce client memory:

Store position map on server. Obliviously!

“Recursive” construction:

Client needs new position map for server-side position map…

Key fact: it is $\alpha$ times smaller!

Repeat this recursively $\log_\alpha(n)$ times. In the end:

- Client position map becomes size $O(1)$.
- Server stores $\log_\alpha(n)$ position maps, each $\alpha \times$ smaller than last.
- Each lookup, $\log_\alpha(n)$ roundtrips to query each position map.
Efficiency of recursive construction

**Overhead.**
Each lookup, $O(\log n)$ recursive calls, each of size $O(\log^2 n)$.
→ $O(\log^3 n)$ overhead.

**Server memory:** $O(n \log n)$.

**Client memory:** $O(1)$. 
Variants

Original Tree ORAM had more complex eviction strategy and analysis, better efficiency.

Path ORAM:
- Client has a small stash of blocks.
- Blocks are evicted along the same branch as item was read.
- Can use nodes as small as $K = 4$ blocks!
Searchable Encryption
Encrypted search:
- Client stores encrypted database on server.
- Client can perform search queries.
- Privacy of data and queries is retained.
Example: private email storage.

*Dynamic* SSE: also allows update queries.
Searchable Symmetric Encryption

Two databases:

- **Document** database:
  
  Encrypted documents $d_i$ for $i \leq D$.

- (Reverse) **index** database $DB$:
  
  Pairs $(w,i)$ for each keyword $w$ and each document index $i$ such that $d_i$ contains $w$.

  $$DB = \{(w,i) : w \in d_i\}$$
A simple solution

Put everything into ORAM.

› Secure.
  (Up to leaking lengths of answers.)

› Inefficient.
  (In certain cases, such as Enron email dataset or English Wikipedia, some studies suggest that a trivial ORAM would be most efficient.)

How to capture leakage

**ORAM security:** accesses can be simulated by a simulator knowing only the **number of accesses**.

- **Formally:** secure iff there exists a simulator, which on input number of accesses, outputs a set of accesses indistinguishable from real algorithm.

**Searchable encryption security:** accesses can be simulated by a simulator knowing only the **output of a leakage function** $L$.

- **Formally:** secure iff there exists a simulator, which on input the output of the leakage function, outputs a set of accesses indistinguishable from real algorithm.

(Leakage function takes as input the database and all operations.)
Security Model

Real world

Client \rightarrow Adversary \rightarrow Server

Query q

Ideal world

L \rightarrow Adversary \rightarrow Simulator

L(q, DB)
Naive approach

For each keyword, just store the encrypted list of matching documents.

\[ w_0: \quad i_0 \quad i_1 \quad i_2 \quad \ldots \quad i_c \]

\[ w_1: \quad i'_0 \quad i'_1 \quad i'_2 \quad \ldots \quad i'_c \]

*Problem:* easy to see number of matching documents for each keyword.
Random approach

Solution: store indices in random places in memory.

Problem: easy to see when new document matches old keyword.
Achieving Forward Security

Solution: use public key encryption.

Forward Security: update operations now leak nothing!
Leakage function for updates is $\bot$. 
Higher practical efficiency

How to do the same thing using only symmetric primitives:
L_0, ..., L_k can be described using log k hash inputs.

For k=2: