Oblivious RAM
Generalizing

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

**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.

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

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

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

<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:

- **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...

1. $n$ real items (some outdated), $s$ dummies
2. Oblivious sort with $\pi^{-1}$
3. $n$ real items (some duplicates)
4. Erase outdated duplicates
5. $n$ real items + some empty
6. Oblivious sort with $\pi'$
7. $n+s$ items sorted with $\pi'$

main memory | stash
---|---
real items, empty items | 

$n$ real items (some duplicates) | $s$ dummies | empty items

$n$ real items (some outdated), $s$ dummies | real items, empty items
Security

**Setup:** server sees:

- main memory
  - $n+s$ encrypted items

- stash
  - $s$ enc. items

**Lookup:** server sees:

- main memory

- stash
  - access to uniformly random fresh location
  - 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):

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

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 $n \geq \lambda$. 
Hierarchical ORAM
A lower bound

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

Proof under assumptions:
- Client memory $O(1)$.
- Statistically secure ORAM.
- “Balls and bins” model.

(Exist stronger proof, Larsen & Nielsen ’18.)
Proof sketch

Each item \( i \) = colored ball. At start:

\[
\begin{array}{ccccccccc}
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
\end{array}
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & \ldots & n & \ldots & \\
\end{array}
\]

\textbf{Client: } c = O(1) \text{ balls} \hspace{1cm} \textbf{Server: } n \text{ balls } + \text{ extra room}

Suppose client wants to make queries for balls \( b_1, \ldots, b_q \).
\( \rightarrow \) 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 \( \rightarrow \) 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.

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

\[
\Rightarrow \quad f(q) = q \cdot \Omega(\log n)
\]
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 \( H \).

![Diagram showing items mapped to buckets via hash function]

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.
Reshuffling

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
- \( \ldots \)

Lookup: server sees:

- size 2
- size 4
- size 8
- \( \ldots \)

- 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.
“Bucket” hashing had total storage $O(n \log n)$, and lookup $O(\log n)$. Cuckoo hashing has storage $(2+\varepsilon)n = O(n)$, 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 $H_2(i)$. 

![Diagram showing Cuckoo hashing with items and hashes]

$m=O(n)$ cells

1 2 3 ... n
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 $i = \text{edge } H_1(i) - H_2(i)$. 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.
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.
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 $O(1)$.

→ 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.

*Tree ORAM*

by Shi et al. ’11
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.)
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 $\log (\frac{n}{\alpha})$. 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 $K$.

→ For one of the two children of $x$, number of blocks whose Pos is below that child $c$ must be at least $K/2$.

→ This implies event \([\text{Pos of new block is below } c]\) happens $K/2$ times, without event \([\text{eviction branch includes } c]\) 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 $\text{Pos}: [1,n/\alpha] \to [1,n/\alpha]$, 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
Outsourcing storage, with search

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 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 \text{Adversary} \rightarrow \text{Server}

Adversary

Query q

Ideal world

L \rightarrow \text{Adversary} \rightarrow \text{Simulator}

L(q, DB)
Naive approach

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

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

\[ w_1: \quad i'_0 \quad i'_1 \quad i'_2 \quad i'_{c'} \]

... 

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

Solution: store indices in random places in memory.

$w_0$: $L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow \ldots \rightarrow L_c$

$w_1$: $L'_0 \rightarrow L'_1 \rightarrow L'_2 \rightarrow \ldots \rightarrow L'_c$

$w$: $L \rightarrow L' \rightarrow \ldots \rightarrow L_c$

$w'$: $L' \rightarrow L'' \rightarrow \ldots \rightarrow L'_c$

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:
Example

$L_0, \ldots, L_k$ can be described using $\log k$ hash inputs.

For $k=2$:
Understanding leakage
Problem Statement

What can the server learn from the above leakage?
Example from earlier

Query A matches records a, b, c.
Query B matches records b, c, d.
Query C matches records c, d.

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

Challenges:

› How do we extract order information? (What algorithm?)
› How do we quantify and analyze how fast order is learned as more queries are observed?
Short answer: there is already an algorithm!

Long answer: **PQ-trees**.

$X$: linearly ordered set. Order is unknown.

You are given a set $S$ containing some intervals in $X$.

A **PQ tree** is a compact (linear in $|X|$) representation of the set of all permutations of $X$ that are compatible with $S$.

Can be updated in linear time.
Full Order Reconstruction

We want to quantify order learning...
Challenge 2: Quantify Order Learning

\[ P \rightarrow Q \]

No information \[ \cdots r_1 r_2 r_3 \cdots \] Full reconstruction \[ \cdots r_1 r_2 r_3 \cdots \]

\(\varepsilon\)-Approximate order reconstruction.

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

No information

Full reconstruction

#queries?

Diameter $\leq \varepsilon N$

$\varepsilon$-Approximate reconstruction
Application of VC theory

No information

$O(N \log N)$ queries

Full reconstruction

$O(\varepsilon^{-1} \log \varepsilon^{-1})$ queries

$\varepsilon$-Approximate reconstruction

Conclusion: learn approximate order in constant time…

Note: some (weak) assumptions are swept under the rug.
In practice

**APPROXORDER** experimental results

$R = 1000$, compared to theoretical $\epsilon$-net bound

<table>
<thead>
<tr>
<th>Number of queries</th>
<th>Max. sacrificed symmetric value $N$</th>
<th>Max. bucket diameter $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>10000</td>
<td>100000</td>
<td>100000</td>
</tr>
</tbody>
</table>

$\leq \varepsilon^{-1} \log \varepsilon^{-1}$

ApproxOrder experimental results

$R = 1000$, compared to theoretical $\epsilon$-net bound
Connection with Machine Learning

› This result used VC theory.
› Other results: known query setting = PAC learning.
› Some results for general query classes.

Machine learning in crypto: also used for side channel attacks. Same general setting!

Natural connection between reconstructing secret information from leakage and machine learning.

 Seems to be a powerful tool to understand the security implications of leakage. In side channels - use learning algorithms; here - use learning theory.
Volume Leakage
Problem Statement

Attacker only sees volumes = number of records matching each query.

What can the server learn from the above leakage?
The attacker wants to learn exact counts.

Some volumes

A volume = number of records matching some range.
KKNO16 Volume Attack

Assume **uniform** queries.

**Step 1:** recover exact probability of every volume $\rightarrow$ number of queries that have each volume.

**Step 2:** express and solve equation system linking above data back to DB counts. (Ends up as polynomial factorization.)

After $O(N^4 \log N)$ uniform queries, previous alg. recovers all DB counts.

Remarks:

- Requires **uniform** distribution.

- **Expensive.** In fact, uses up *all possible* leakage information!

- Lower bound of $\Omega(N^4)$. 
Elementary Volumes

Counts

Value

3

7

1

12

1

2

3

4

“Elementary” ranges

Elementary volumes = volumes of ranges [1,1], [1,2], [1,3]...
Elementary Volumes

Counts  3  7  1  12

Value  1  2  3  4

Fact:

\[ \text{vol}([a,b]) = \text{vol}([1,b]) - \text{vol}([1,a]) \]

so...

- Every volume is = difference of two elementary volumes.
- Knowing set of elementary volumes \(\iff\) knowing counts.

Our goal: finding elementary volumes.
Assumption: the volumes of all queries are observed.

Draw an \textbf{edge} between volumes $a$ and $b$ iff $|b-a|$ is a volume.
Summary

**Attack:** *elementary volumes* form a clique in the volume graph $\rightarrow$ clique-finding algorithm reveals them.

For structured queries, even just volume leakage can be quite damaging. Attack requires strong assumption.

*Remark:* clique finding can be avoided via pre-processing.