Techniques in Cryptography and Cryptanalysis

Part 2: Oblivious Algorithms

Brice Minaud

email: brice.minaud@ens.fr

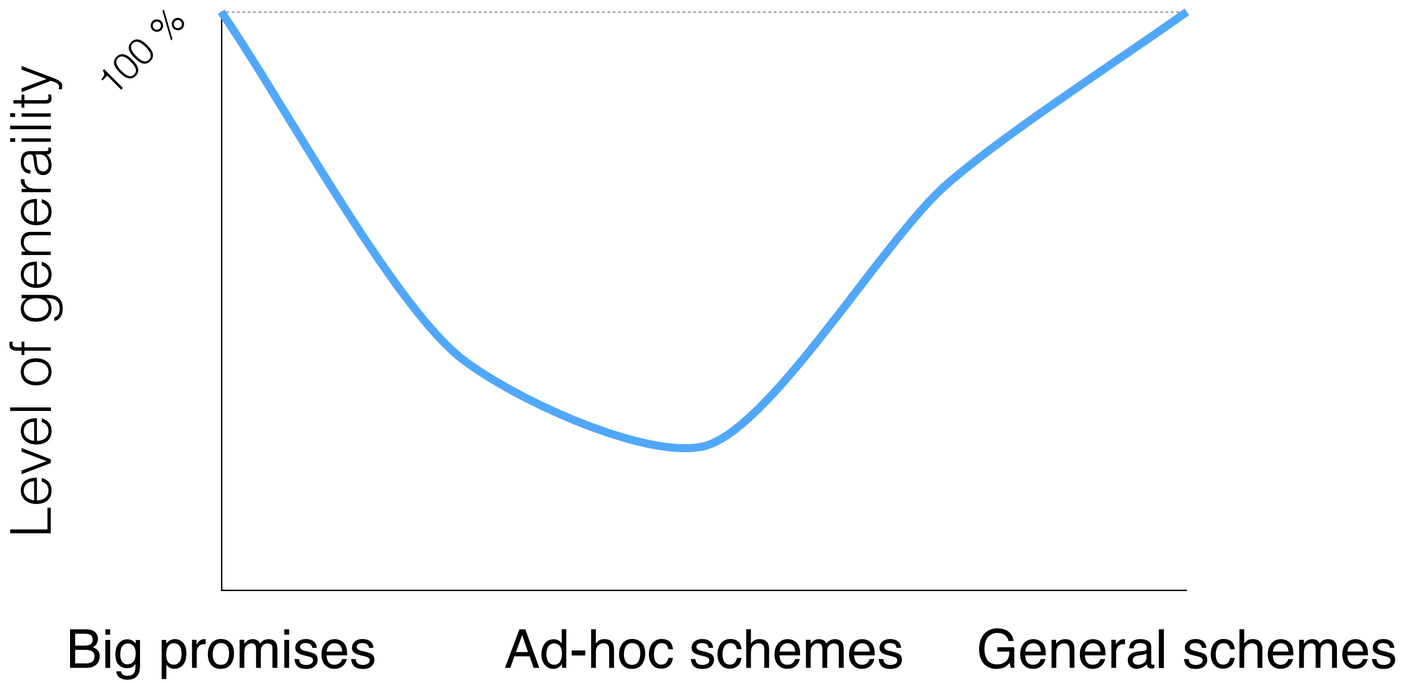
MPRI, 2023

Meta information

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

- Zero-knowledge proofs	✔ done
- Oblivious algorithms	📍 *we are here*
- Lattices	2nd period

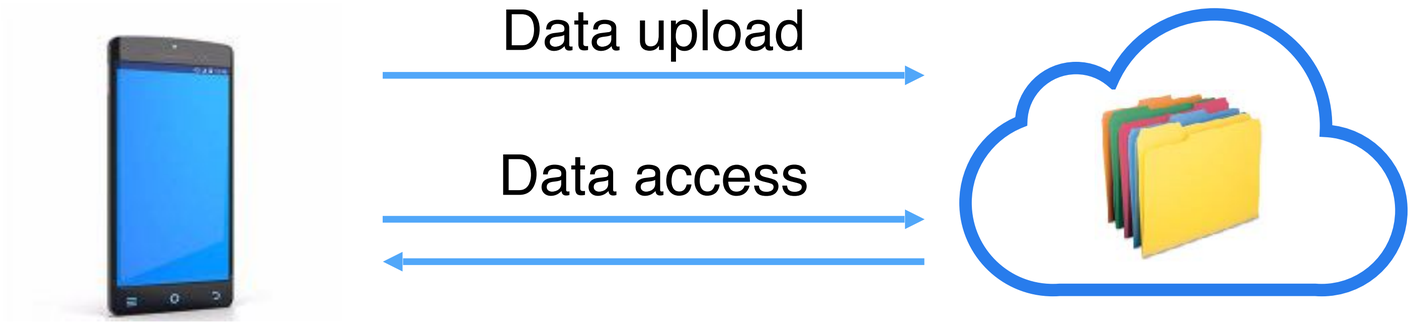
Roadmap



Course contents:

- 1. Motivation (big promises).
- 2. Oblivious sorting.
- 3. Oblivious RAM.

Outsourcing storage

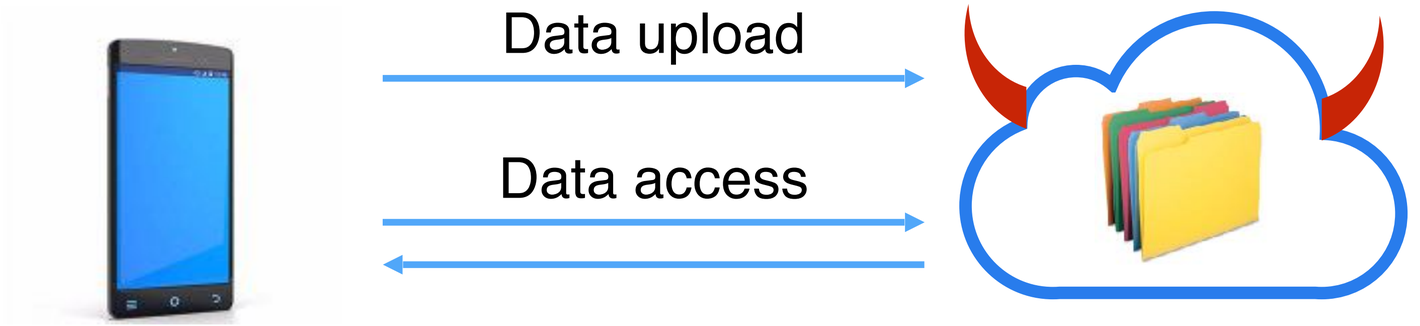


Scenario: Client outsources storage of sensitive data to Server.

Examples:

- Company outsourcing customer/transaction info.
- Private messaging service.
- Trusted processor accessing RAM.

Outsourcing storage



Client

Server = adversary

Scenario: Client outsources storage of sensitive data to Server.

Adversary: **honest-but-curious** server.

Security goal: **privacy** of data and queries.

Privacy of data: just encrypt?

Symmetric Encryption.

Message space **M**, ciphertext space **C**, key space **K**.

Setup: Pick key $K \leftarrow_{\$} \mathbf{K}$.

Encryption: encryption of $M \in \mathbf{M}$ is $C = \mathrm{Enc}_K(M) \in \mathbf{C}$.

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

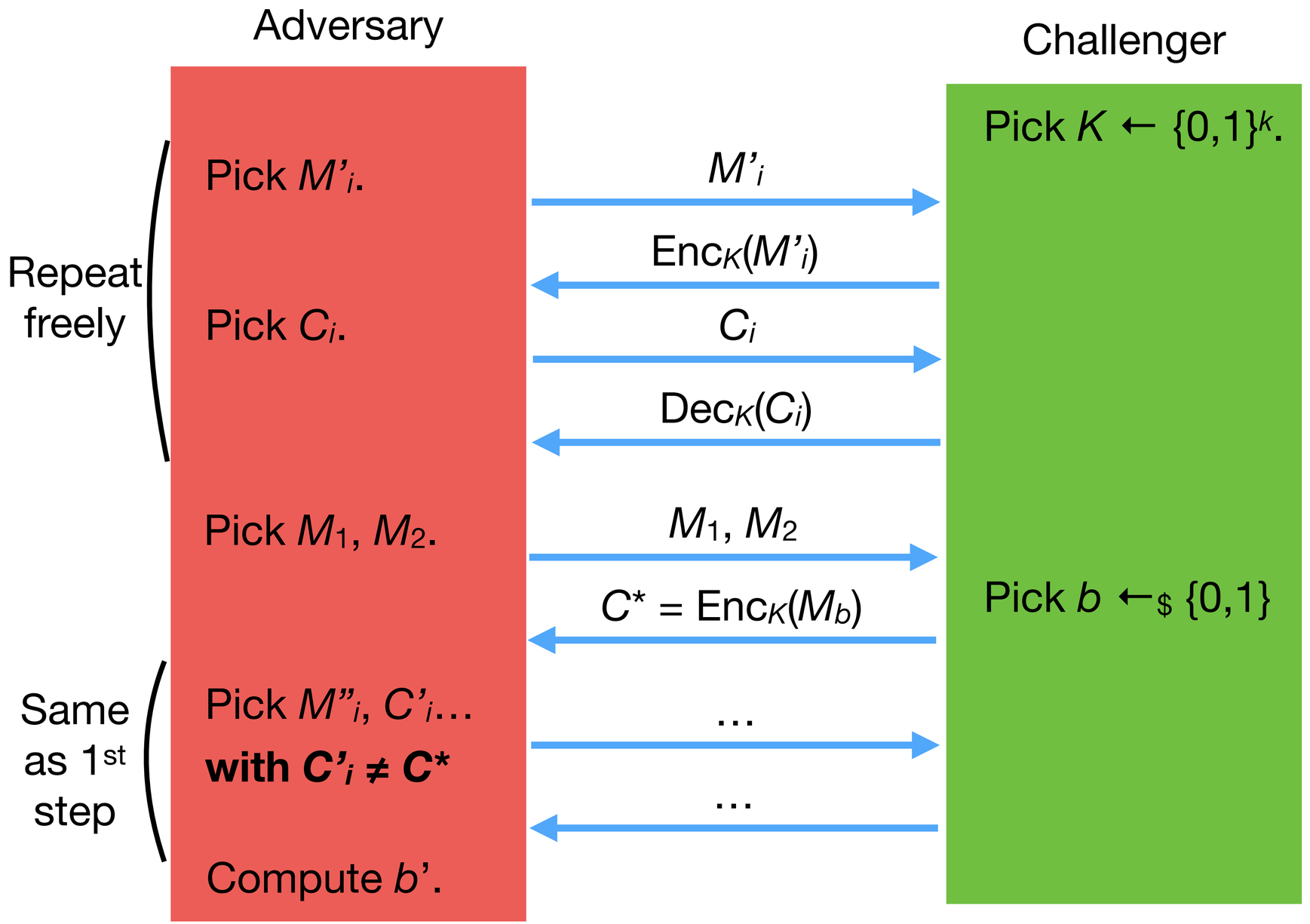
Perfect secrecy (sketch):

Let M_0 and M_1 be two arbitrary messages.

Perfect secrecy: $\mathrm{Enc}_K(M_0) = \mathrm{Enc}_K(M_1)$.

The equality is an equality of distributions. The randomness is over the uniform choice of K.

IND-CCA: indistinguishability under Chosen-Ciphertext Attacks



Symmetric encryption: “proper” definition

Symmetric Encryption.

Message space **M**, ciphertext space **C**, key space **K**.

Setup: Pick key $K \leftarrow_{\$} \mathbf{K}$.

Encryption: encryption of $M \in \mathbf{M}$ is $C = \mathrm{Enc}_K(M) \in \mathbf{C}$.

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

Correctness: for all $M \in \mathrm{M}$,

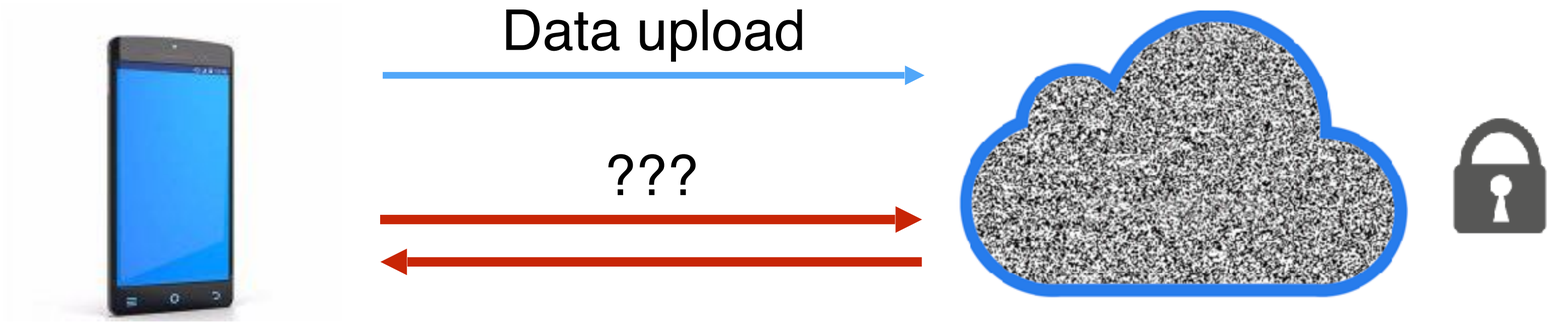
$$\mathrm{Dec}_K(\mathrm{Enc}_K(M)) = M.$$

Security: IND-CCA (for instance).

Caveats:

- Deterministic scheme cannot be IND-CCA. Need randomness, or nonces (→ mode of operation).
- “Security” above only covers **confidentiality**, not **integrity**.

Naive solution



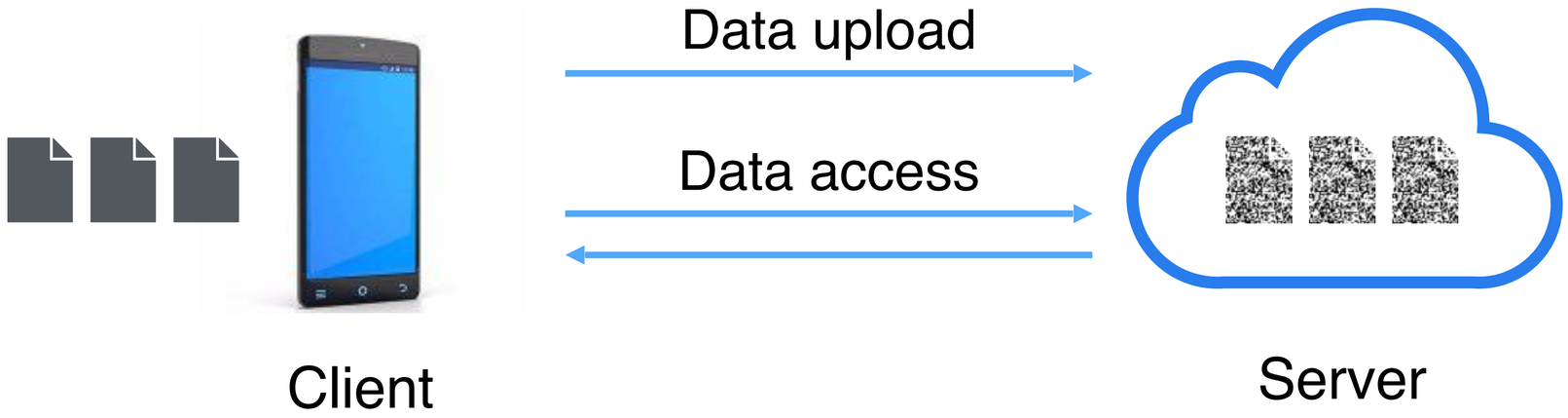
Naive solution: encrypt *all* data using symmetric encryption.

✓ Secure

× Not functional

How can you fetch (or update) specific files?

Naive solution, attempt #2



Naive solution 2: encrypt each file individually.

✓ Functional

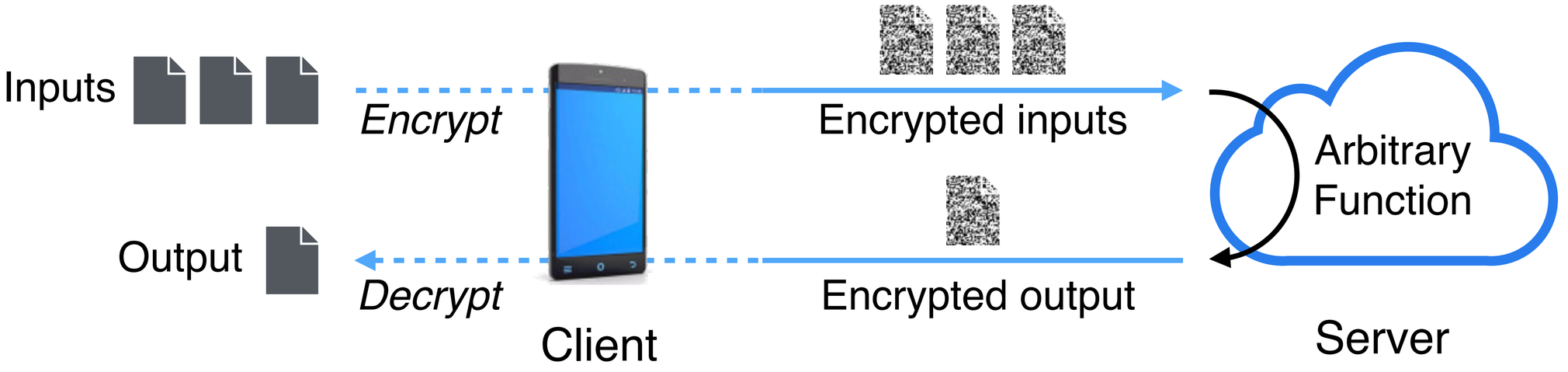
× Not secure

What about security?

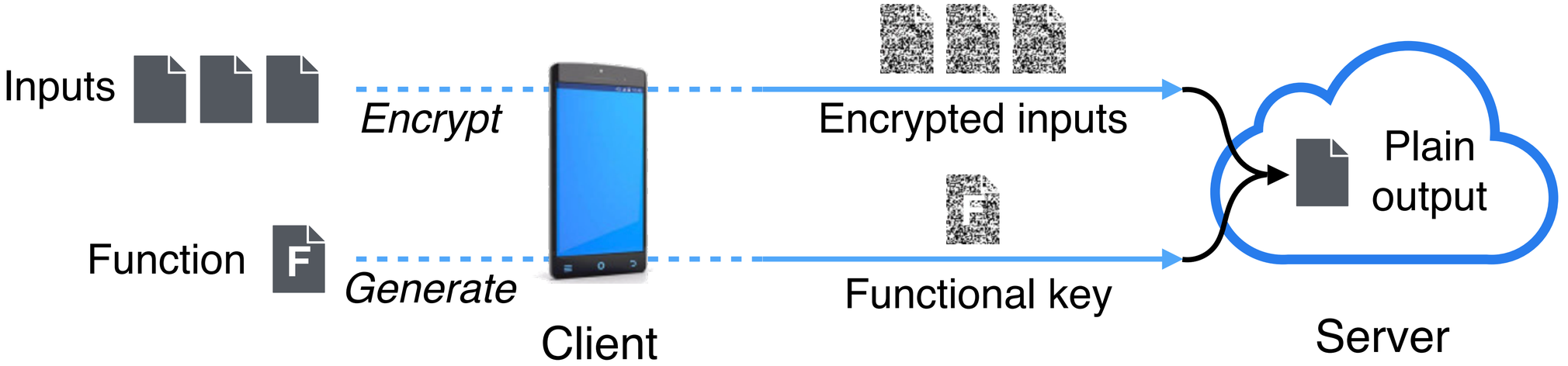
The server learns your **access pattern.**

Some perspective: computing on encrypted data

SNARKs: prove arbitrary statements on encrypted data.



Fully Homomorphic Encryption

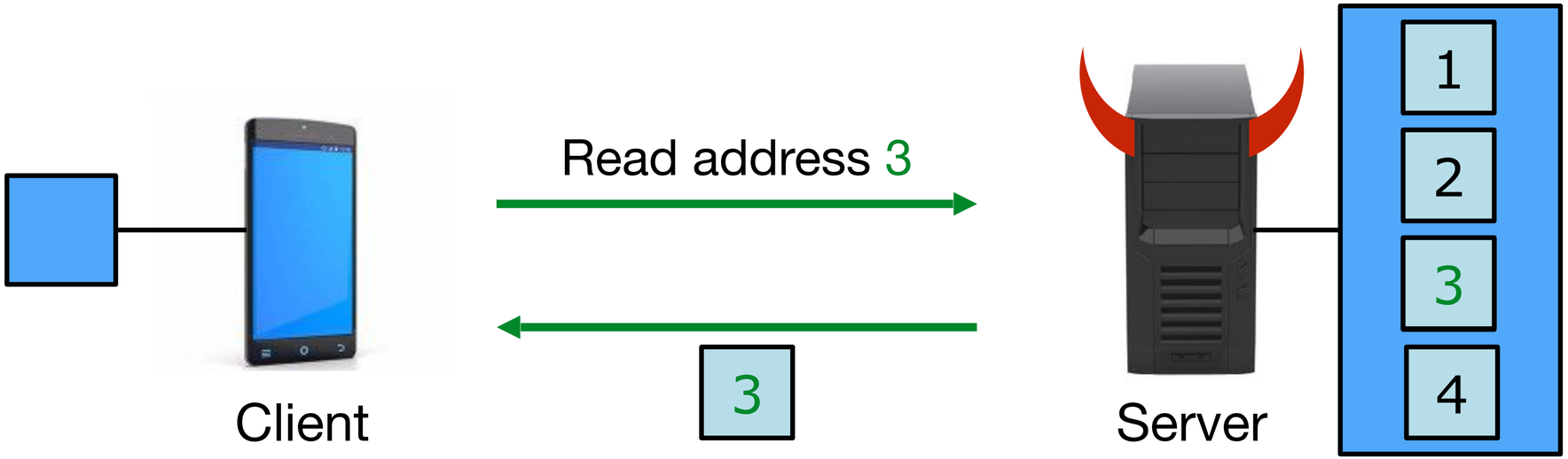


Functional Encryption

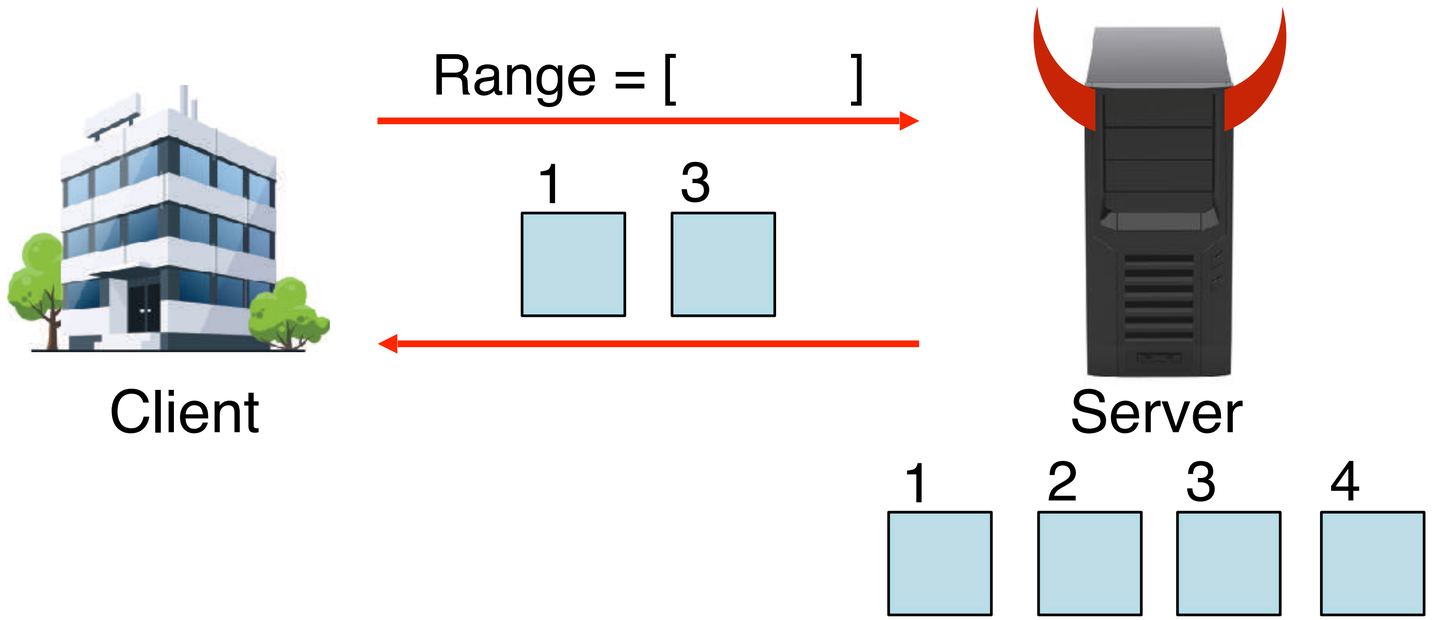
Is it okay to leak access pattern?

(No.)

Does leaking access pattern matter?



Example: range queries



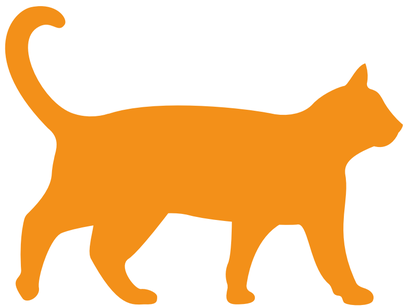
Imagine hospital storing patient information.

Sometimes searches for all patients with ages between *a* and *b*.

What can the server learn from the above leakage?

Connection with **machine learning.**

VC Theory



C

VC Theory

Foundational paper: **Vapnik and Chervonenkis, 1971**.

Uniform convergence result.

Now a foundation of learning theory, especially PAC (*probably approximately correct*) learning.

Wide applicability.

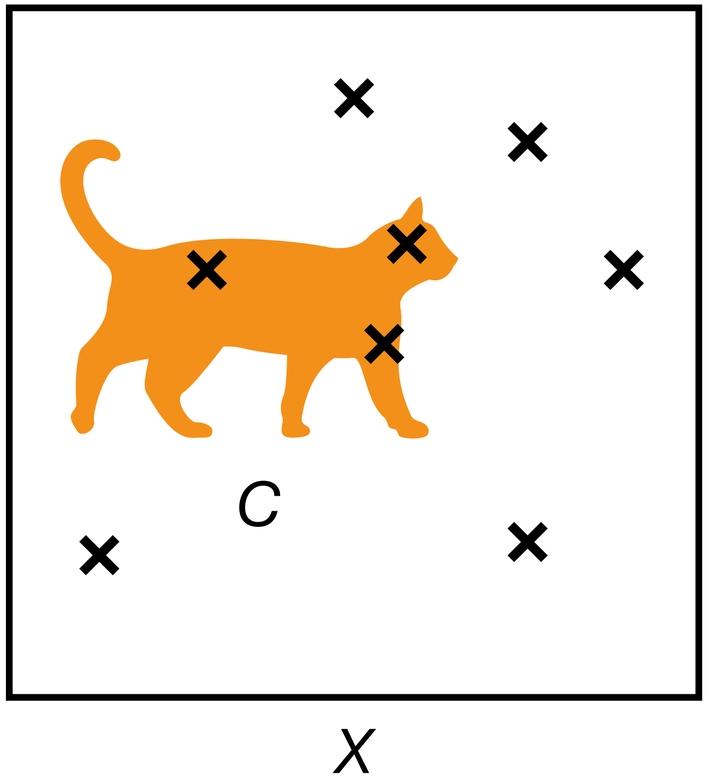
Fairly easy to state/use.

(You don't have to read the original article in Russian.)

Warm-up

Set X with probability distribution D.

Let $C \subseteq X$. Call it a *concept*.



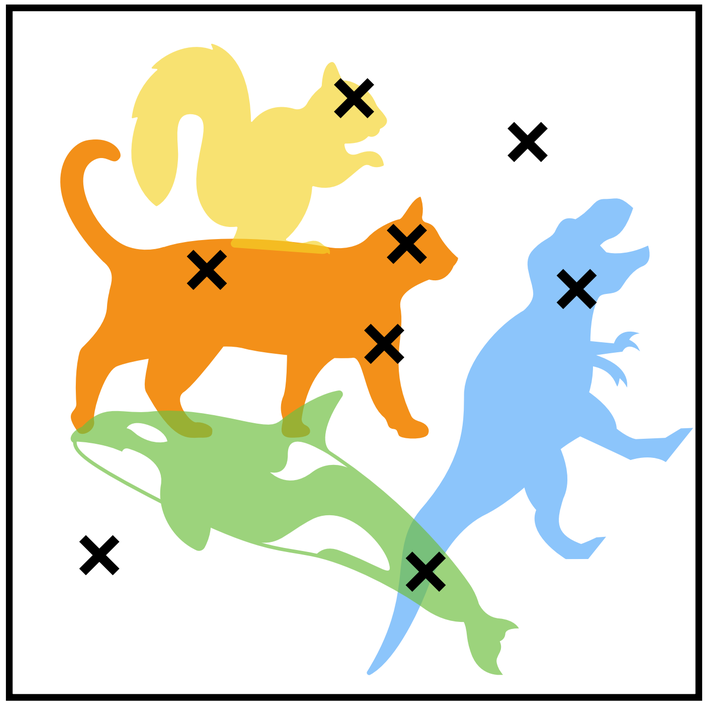
$$\Pr(C) \approx \frac{\#\text{points in } C}{\#\text{points total}}$$

Sample complexity:
to measure Pr(*C*) within ε,
you need $O(1/\varepsilon^2)$ samples.

Approximating a Concept Set

Now: set $\mathcal{C}$ of concepts.

Goal: approximate their probabilities *simultaneously*.



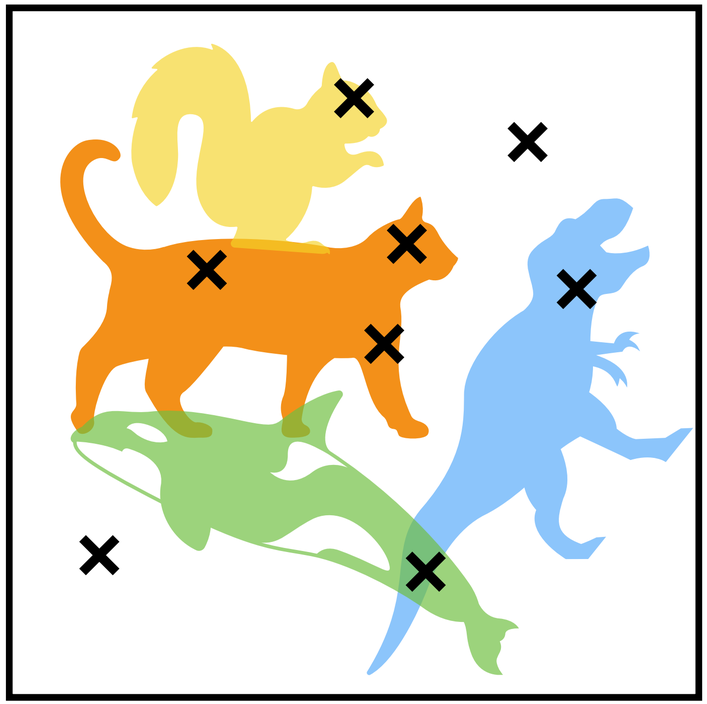
X

The set of samples drawn from X is an **ε-sample** iff for **all** C in $\mathcal{C}$:

$$\left| \Pr(C) - \frac{\#\text{points in } C}{\#\text{points total}} \right| \leq \epsilon$$

ε-sample Theorem

How many samples do we need to get an ε-sample whp?



X

Union bound: yields a sample complexity that depends on $|\mathcal{C}|$.

V & C 1971:

If $\mathcal{C}$ has **VC dimension** d, then the number of points to get an ε-sample whp is

$$O(\frac{d}{\epsilon^2} \log \frac{d}{\epsilon}).$$

Does not depend on $|\mathcal{C}|$!

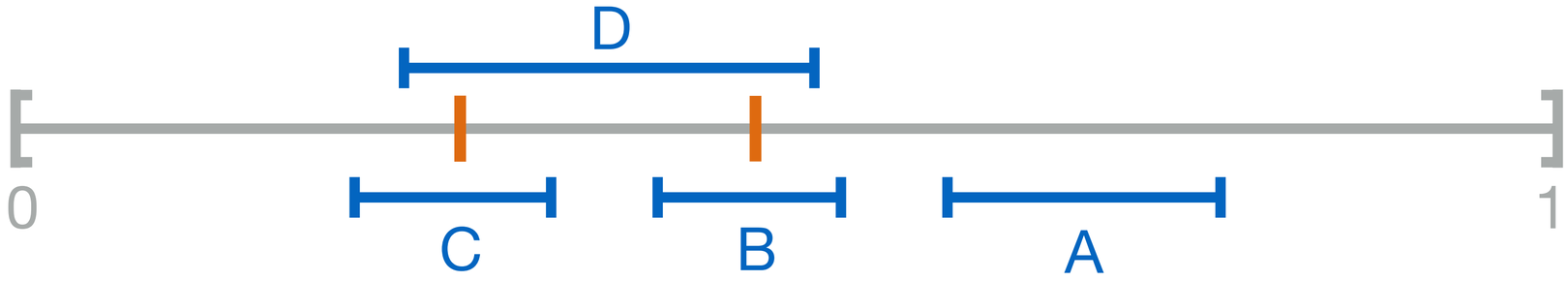
VC Dimension

Remaining Q: *what is the VC dimension?*

A set of points is **shattered** by $\mathcal{C}$ iff:

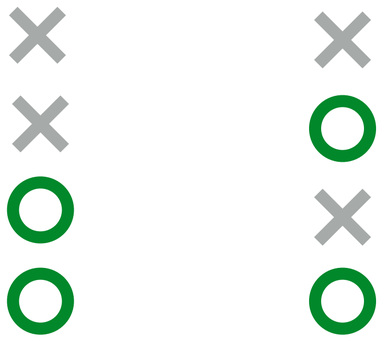
every subset of *S* is equal to *C*∩*S* for some *C* in $\mathcal{C}$.

Example. Take **2 points** in X=[0,1]. Concepts $\mathcal{C}$ = all ranges.



Subsets:

2 points = SHATTERED



OK. Range A.

OK. Range B.

OK. Range C.

OK. Range D.

VC Dimension

Example. Take **3 points** in X=[0,1]. Concepts $\mathcal{C}$ = all ranges.

Subset: ○ × ○ Problem.

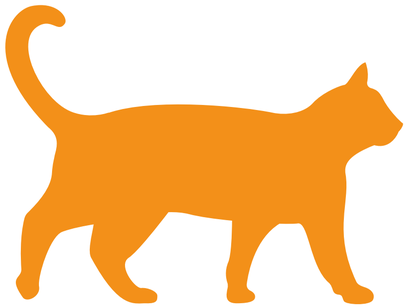
3 points = NOT SHATTERED

VC dimension of $\mathcal{C}$ = largest cardinality of a set of points in *X* that is shattered by $\mathcal{C}$.

E.g. VC dimension of ranges is 2.

What typically matters is just that VC dim is finite.

Statistical Reconstruction

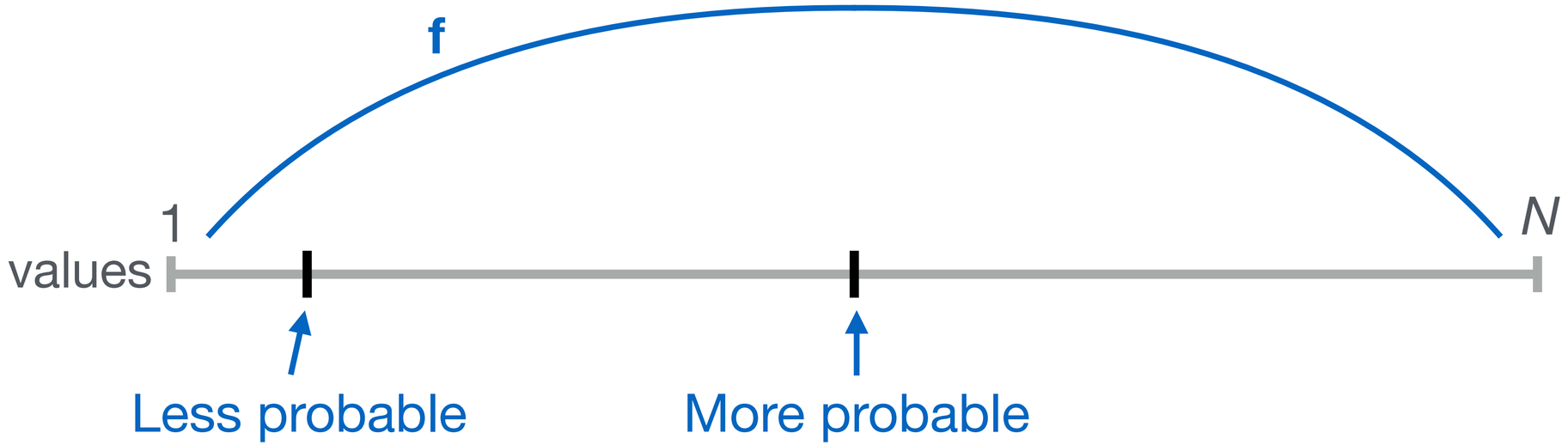


C

vs.

1 3

Direct statistical attack



Assume a **uniform distribution** on range queries.

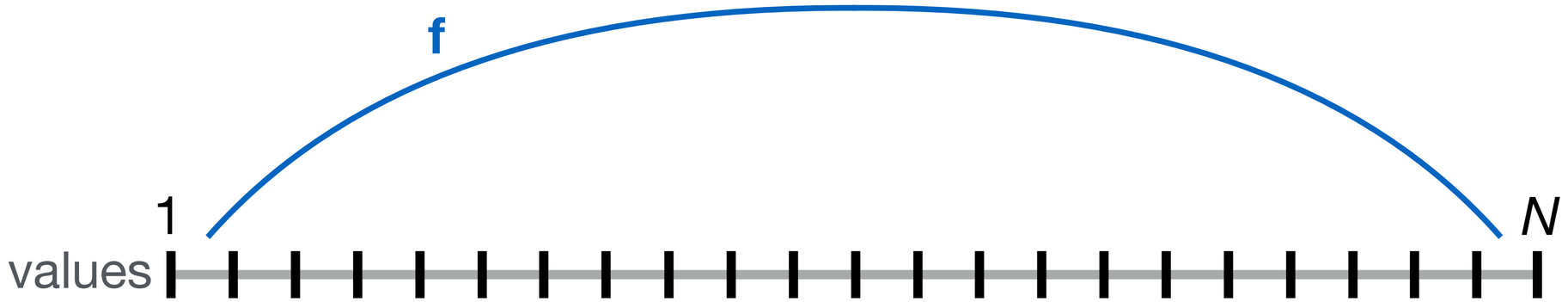
Induces a distribution **f** on the prob. that a given value is hit.

Idea: for each record...

1. Count frequency at which the record is hit.
→ gives estimate of probability it's hit by uniform query.

2. deduce estimate of its value by "inverting" **f**.

Direct statistical attack



Step 1: for **all** records, estimate prob of the record being hit.

This is an ε-sample!

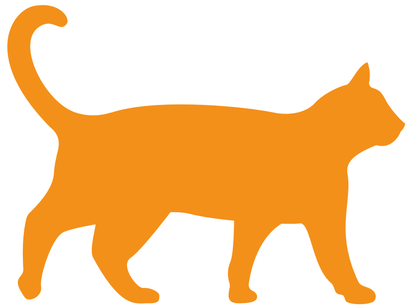
X = ranges $\quad\quad \mathcal{C}$ ={{ranges ∋ x}: x ∈ [1,N]}

so we need $O(\varepsilon^{-2} \log \varepsilon^{-1})$ queries.

Step 2: because **f** is quadratic, “inverting” **f** adds a square.

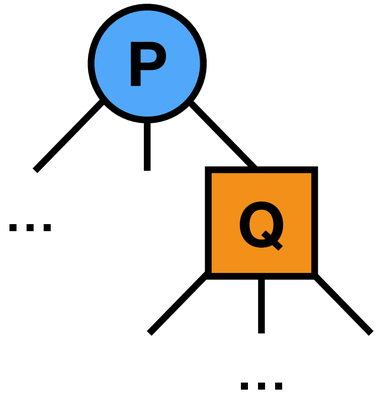
After $O(\varepsilon^{-4} \log \varepsilon^{-1})$ queries, the value of all records is recovered within εN.

Order Reconstruction



C

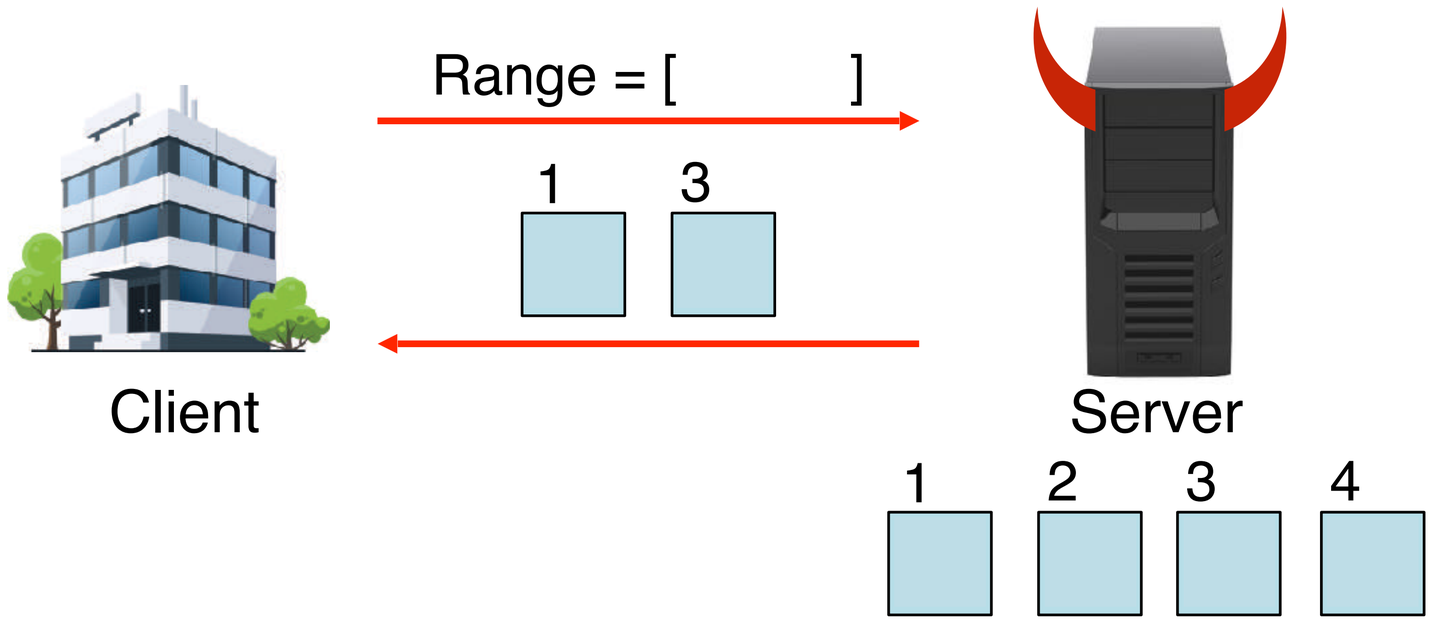
+



...

...

Problem Statement



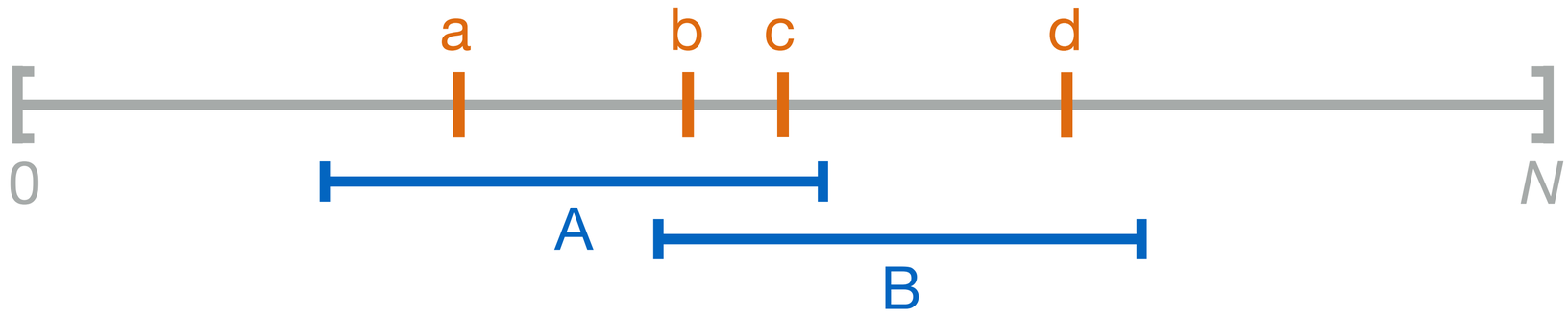
What can the server learn from the above leakage?

This time we **don't assume** i.i.d. queries, or knowledge of their distribution.

Range Query Leakage

Query A matches records a, b, c.

Query B matches records b, c, d.



Then this is the only configuration (up to symmetry)!

→ we learn that records b, c are *between* a and d.

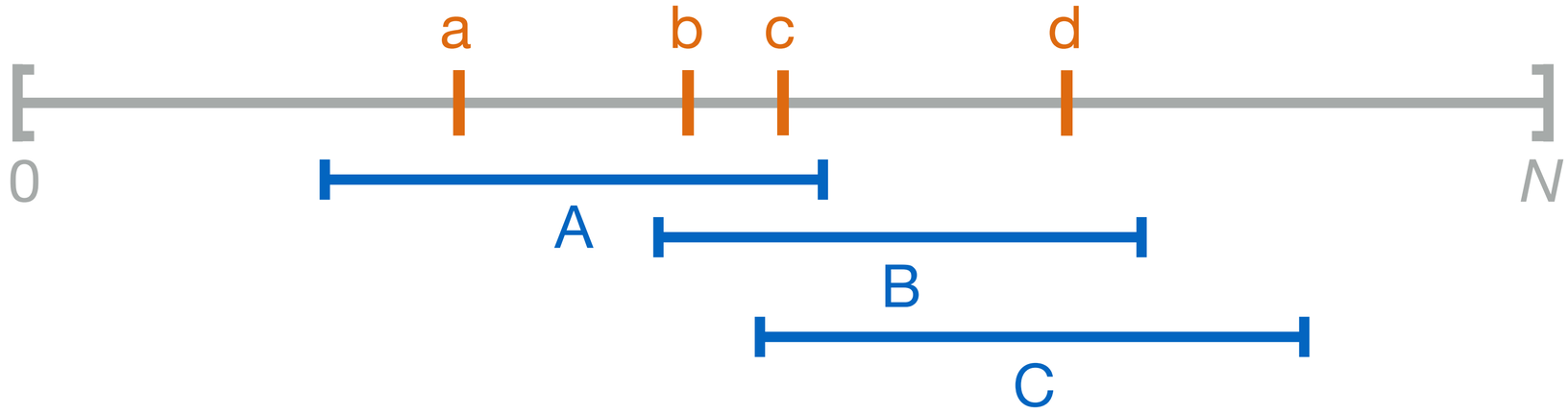
We learn something about the **order** of records.

Range Query Leakage

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?

Challenge 1: the Algorithm

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.

Challenge 2a: quantify order learning

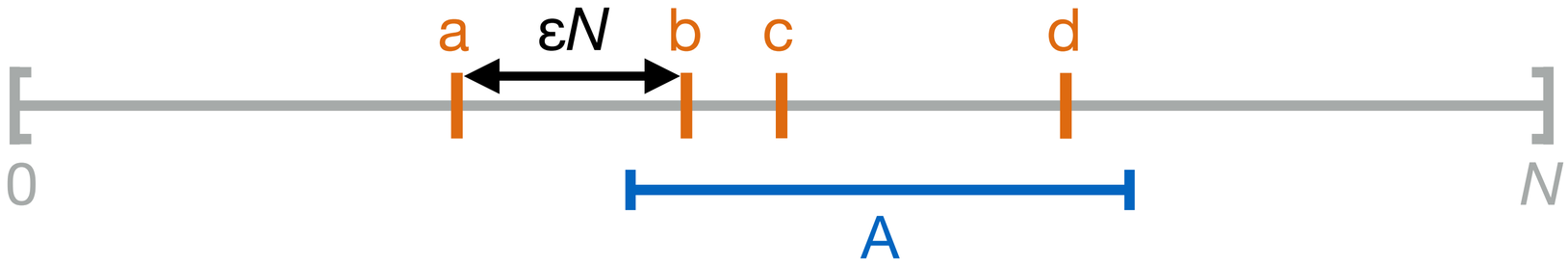
Strongest goal: **full** database reconstruction = recovering the exact value of every record.

More general: **approximate** database reconstruction = recovering all values within εN.

ε = 0.05 is recovery within 5%. ε = 1/N is full recovery.

("Sacrificial" recovery: values very close to 1 and *N* are excluded.)

Challenge 2b: analyze query complexity



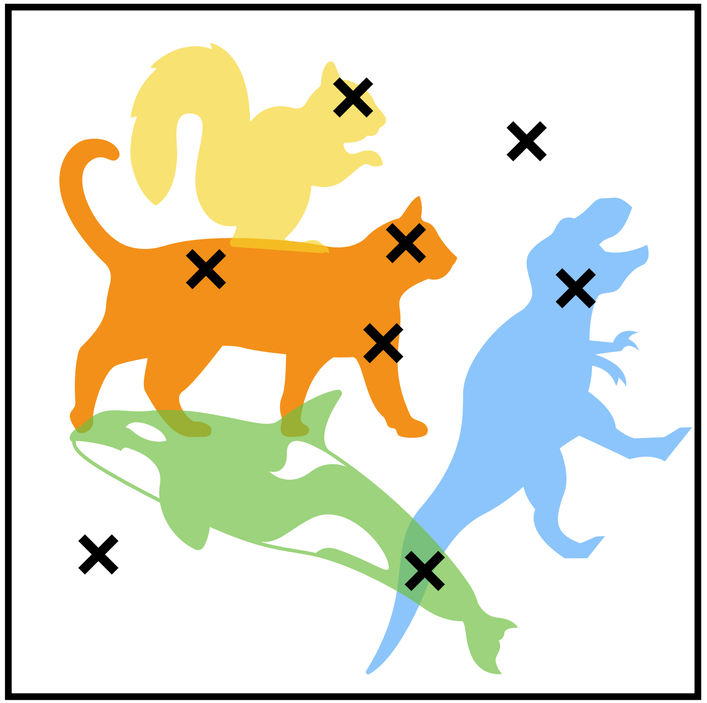
Intuition: if no query has an endpoint between a and b, then a and b can't be separated.

→ ε-approximate reconstruction is impossible.

You want a query endpoint to hit every interval ≥ *εN*.
Conversely with some other conditions it's enough.

Heavy sweeping of details under rug.

VC Theory saves the day (again)



ε-samples: the ratio of points hitting each concept is close to its probability.

What we want now: if a concept has high enough probability, it is hit by at least one point.

The set of samples drawn from X is an **ε-net** iff for all C in $\mathcal{C}$:

$$\Pr(C) \geq \epsilon \Rightarrow C \text{ contains a sample}$$

→ Number of points to get an ε-net whp: $O\left(\frac{d}{\epsilon}\log\frac{d}{\epsilon}\right)$

Access pattern leakage: conclusion

Say patient age has N possible values (e.g. $N = 100$)...

Full order reconstruction: $O(N \log N)$ queries.

Approximate order reconstruction (within εN): $O(\varepsilon^{-1} \log \varepsilon^{-1})$ queries!

(NB: this is optimal.)

Age data: can infer value from order (if all ages are present)...

In this setting, encryption was ultimately useless.

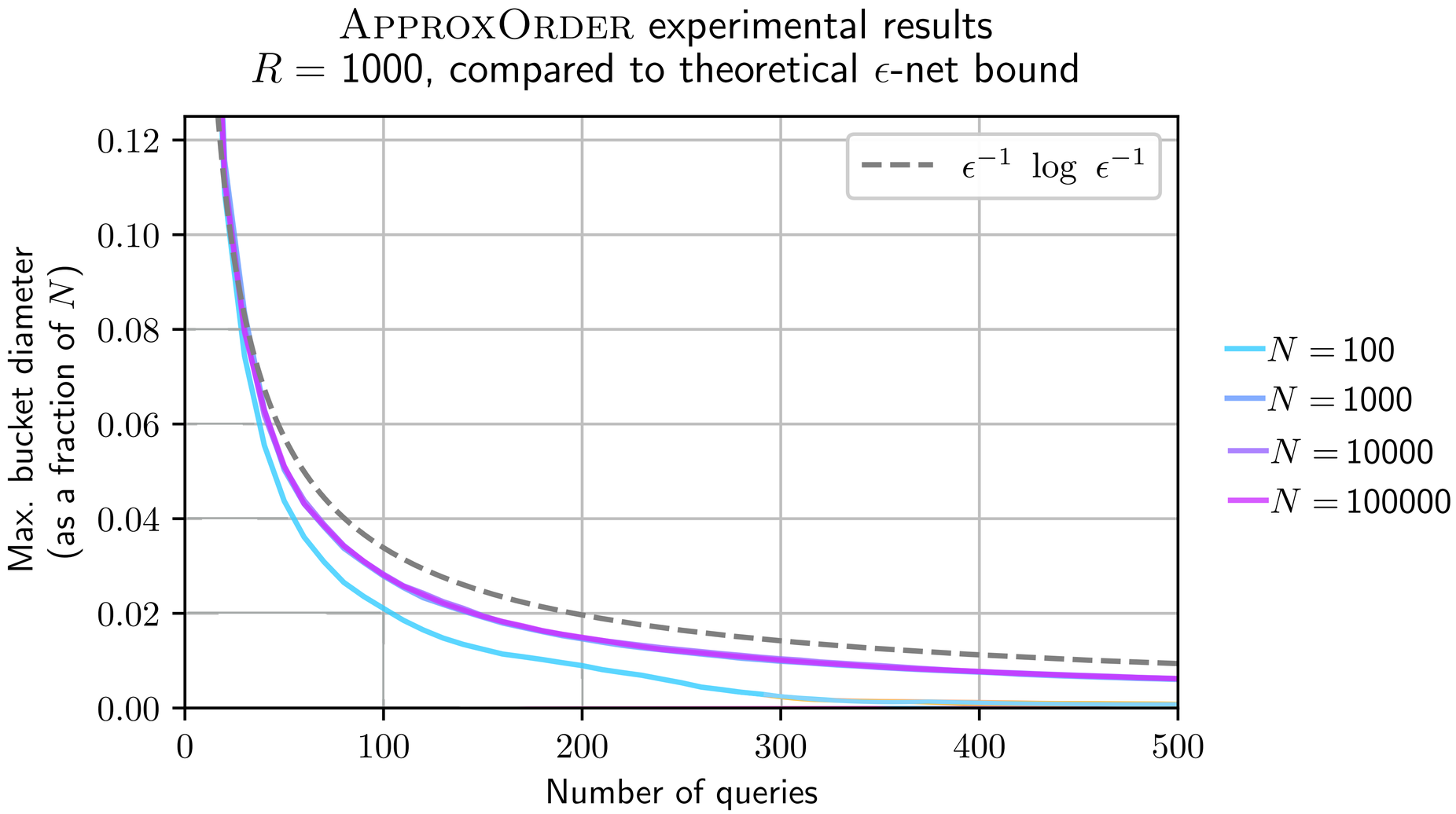
Very rough summary :

highly structured queries

⇒ low VC dimension

⇒ learn data with few queries

It actually works, by the way



Other examples

Suppose you implement AES using lookup tables (for S-boxes).

If adversary can observe queries to tables, AES is **broken**.

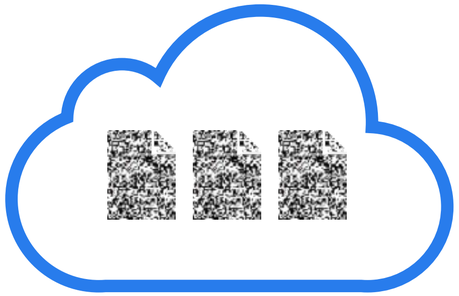
If adversary can observe *cache misses* from access to AES S-box tables, **also broken**.

Two issues:

- Leaking access pattern can be (very) damaging.
- Many settings leak access pattern, completely or partially.

  Cloud storage, trusted enclaves, cache attacks (incl. hypervisors), etc. See also: side-channel attacks.

Oblivious algorithms



Naive solution, attempt #2

Naive solution 2: encrypt each file individually.

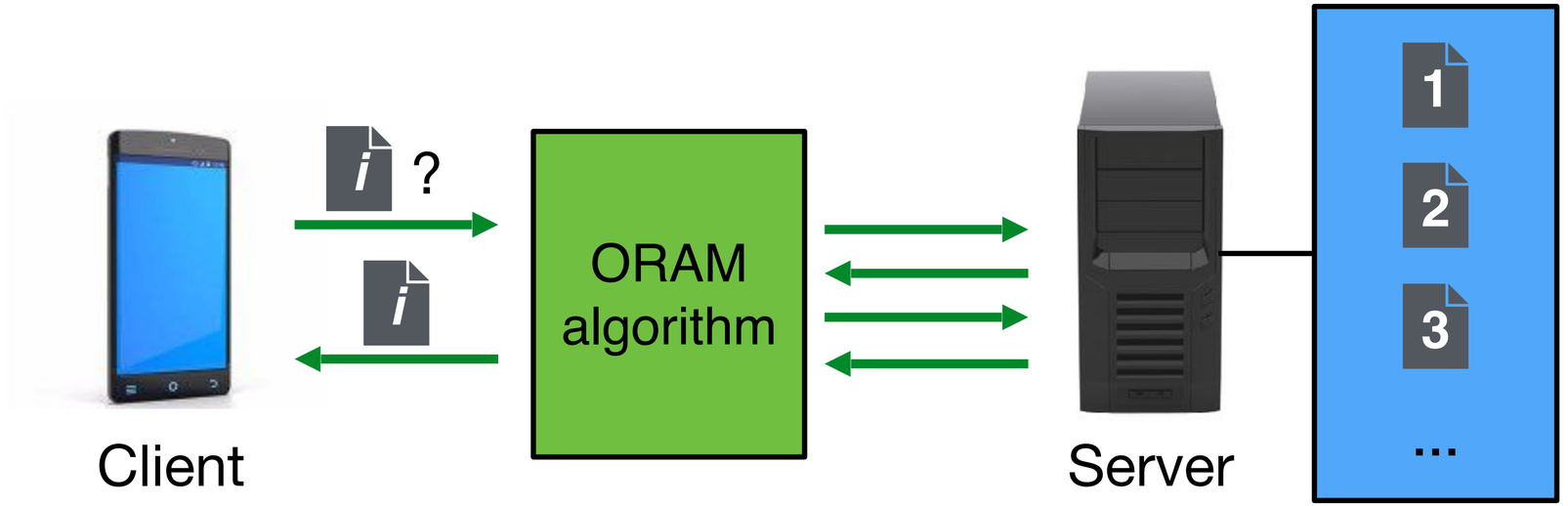
✓ Functional

× **Not secure**

The server learns the client's **access pattern.**

Remark: the “client” could be an algorithm.

Magic Claim



Server stores *N* files.

Client fetches file *i*.

Security: Server learns *nothing* about *i*.

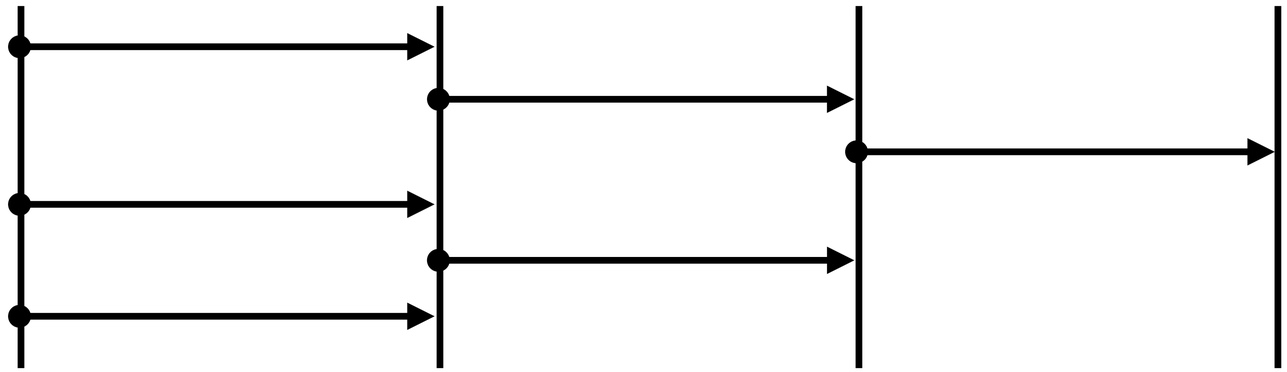
Efficiency: ORAM algorithm only queries O(log *N*) files.

Bonus feature: server performs no computation. Acts like a RAM.

Oblivious algorithm: definition

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.

Oblivious Sorting



Sorting algorithms

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.

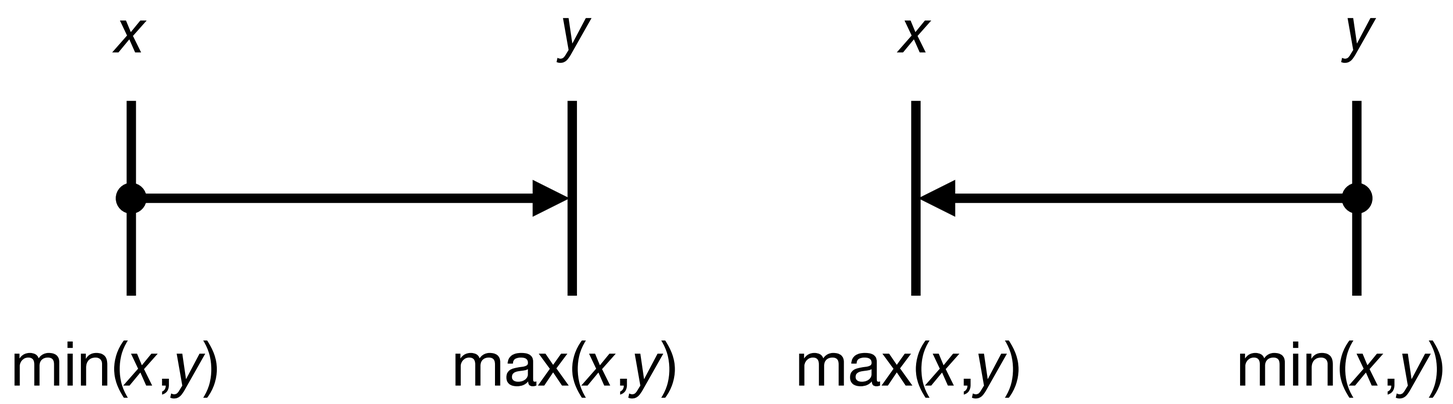
Which of the following algorithms are oblivious?
(assuming inputs are arrays of fixed size.)

1. Bubble Sort. ✔ yes
2. Quick Sort. ✘ no
3. Merge Sort. ✘ no

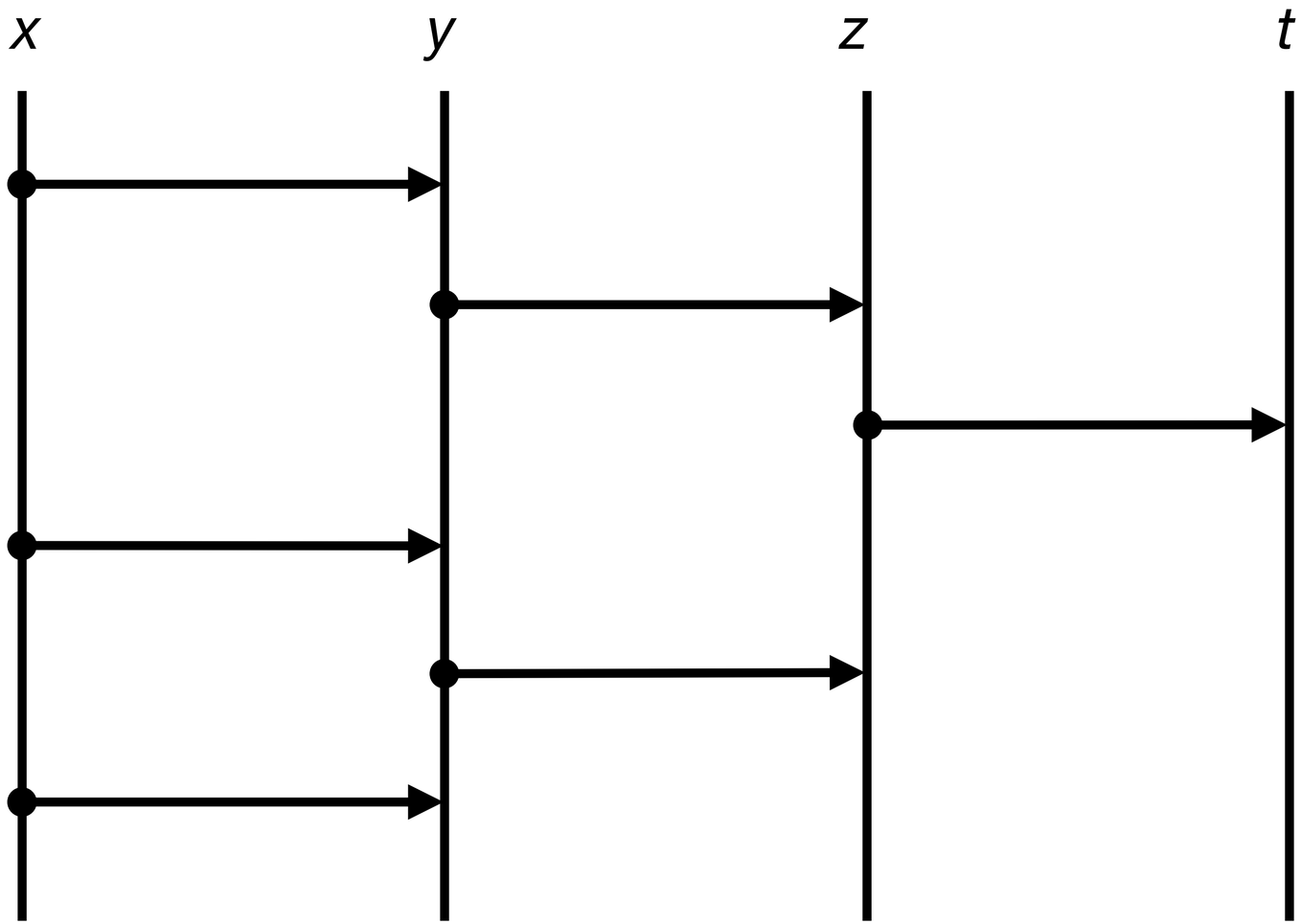
Sorting obliviously

Basic operation: sorting two elements.

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



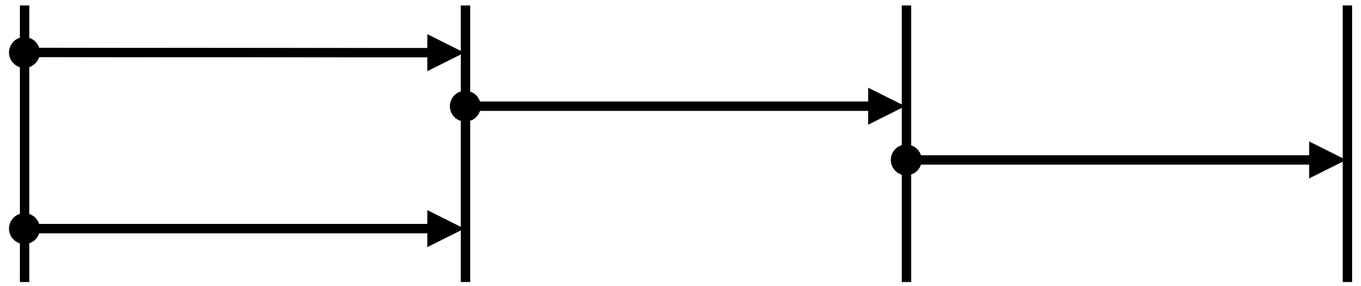
Bubble Sort



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:



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

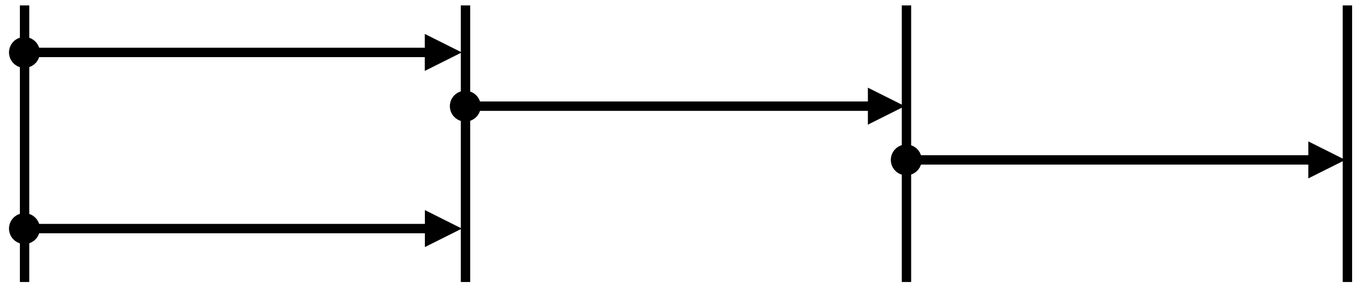
Remark: testing whether a comparator network is a sorting network is co-NP-complete.

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: $x \le y$ implies $f(x) \le f(y)$.

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 $x \le x_i$, 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.

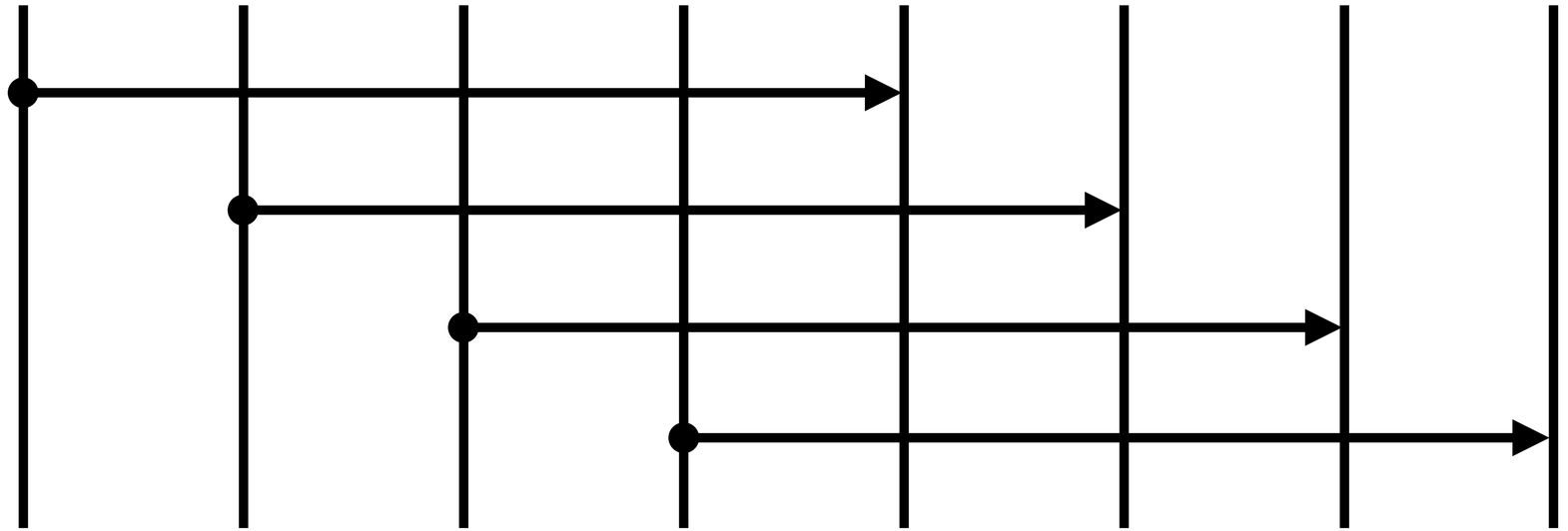
Example: bitonic sequences of 0 and 1's are those of the form $0^a1^b0^c$ and $1^a0^b1^c$.

Half-cleaner

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

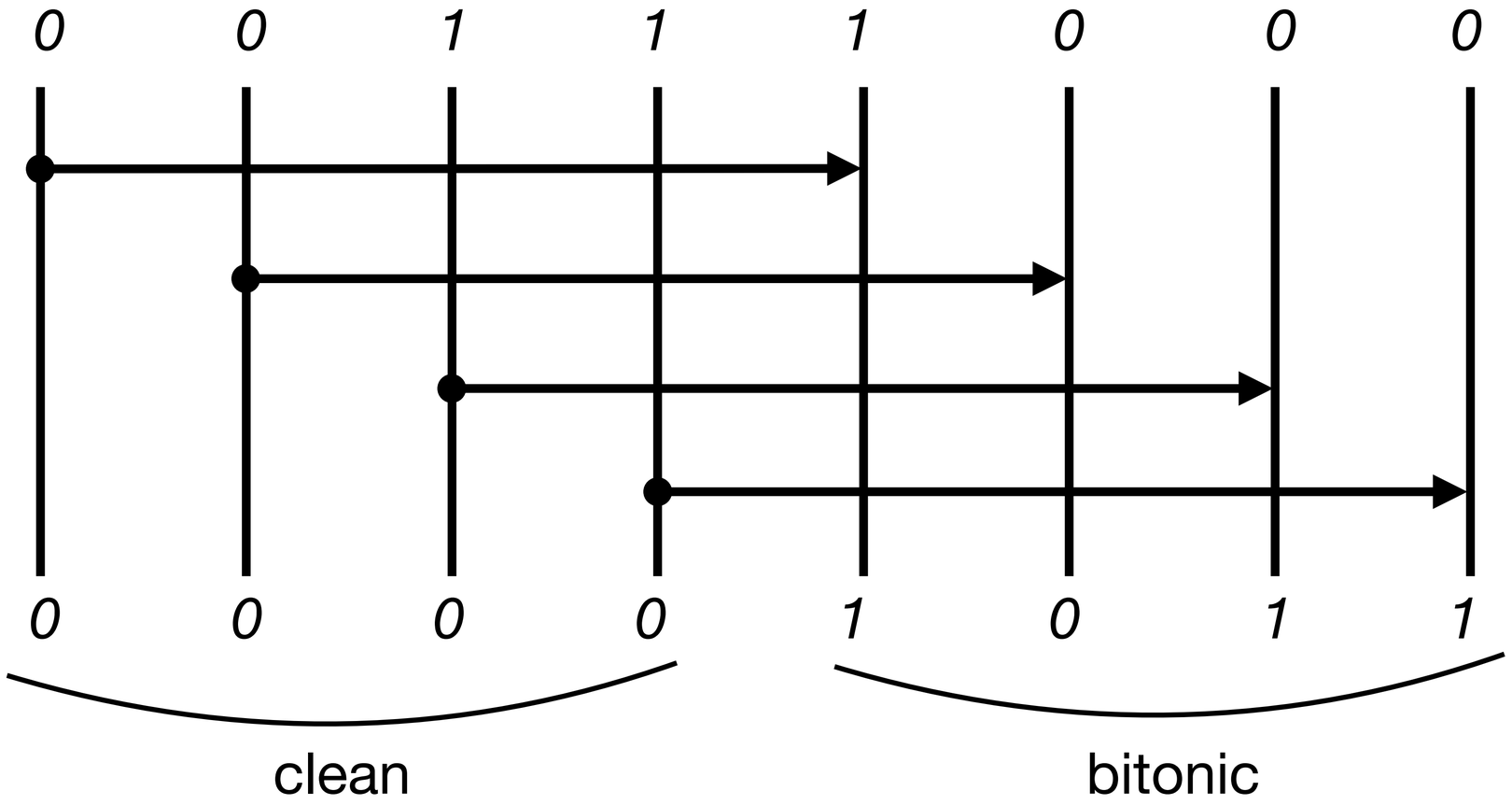
$$(1,n/2+1),\ (2,n/2+2),\ \ldots,\ (n/2,n)$$

Half-cleaner for $n = 8$:



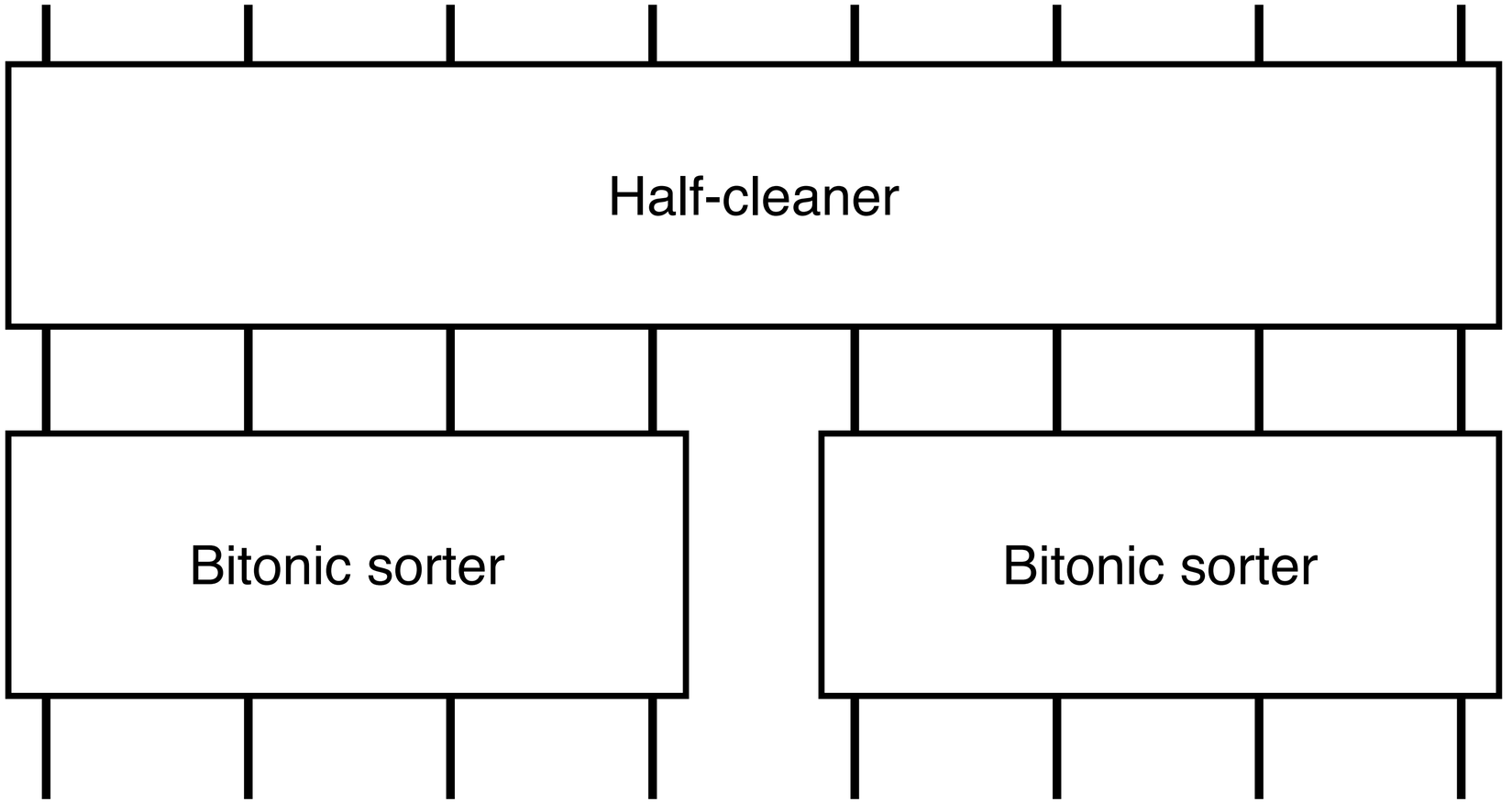
Half-cleaner

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.



Bitonic sorter

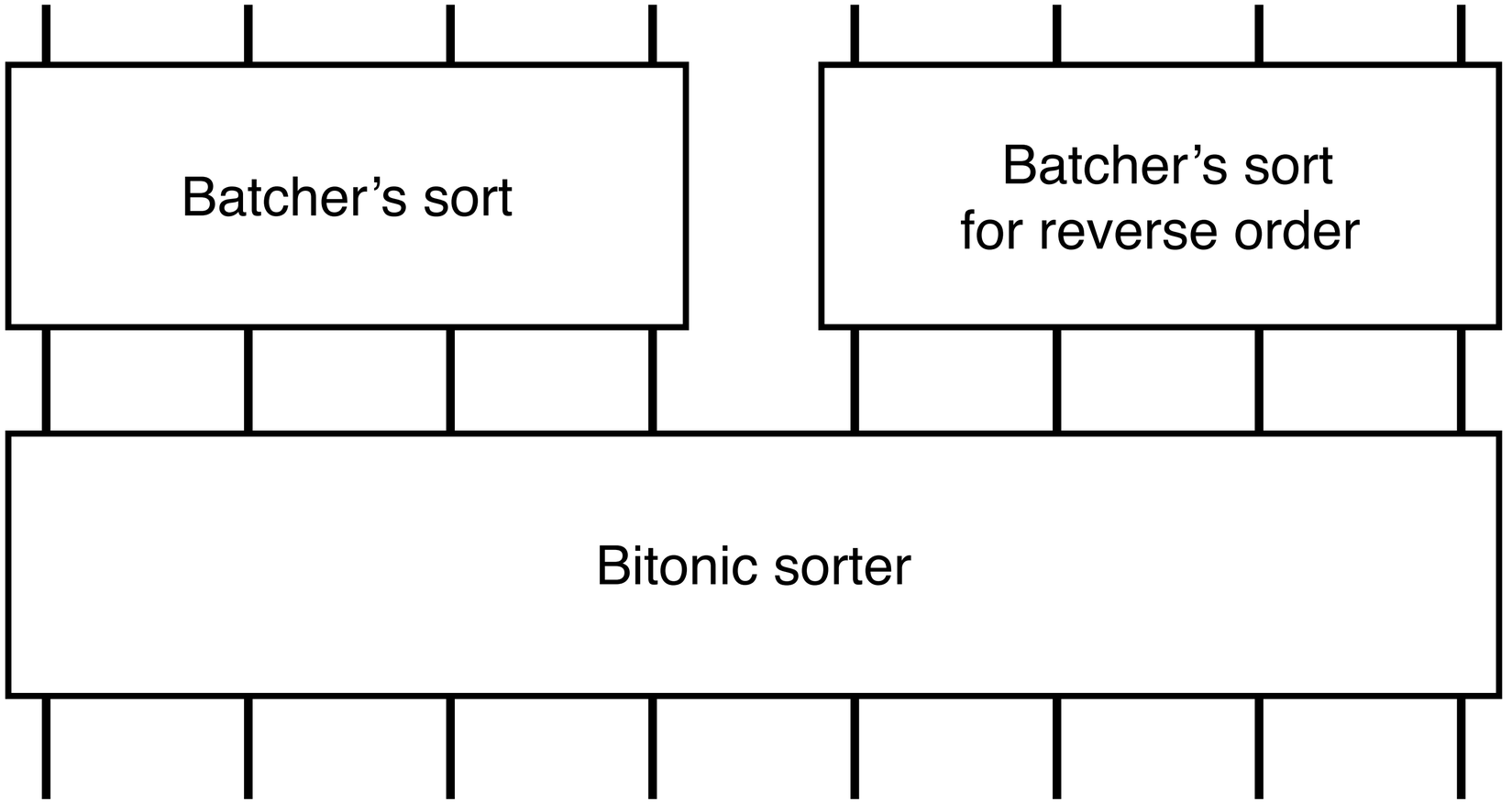
Bitonic sorter recursive construction:



The bitonic sorter correctly sorts all **bitonic** inputs.

Batcher's sort

Batcher's sort recursive construction:



Batcher's sort correctly sorts **all** binary inputs, hence **all** inputs.

Efficiency

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 $B(n) = S(n) + 2B(n/2) = O(n \log^2 n)$.

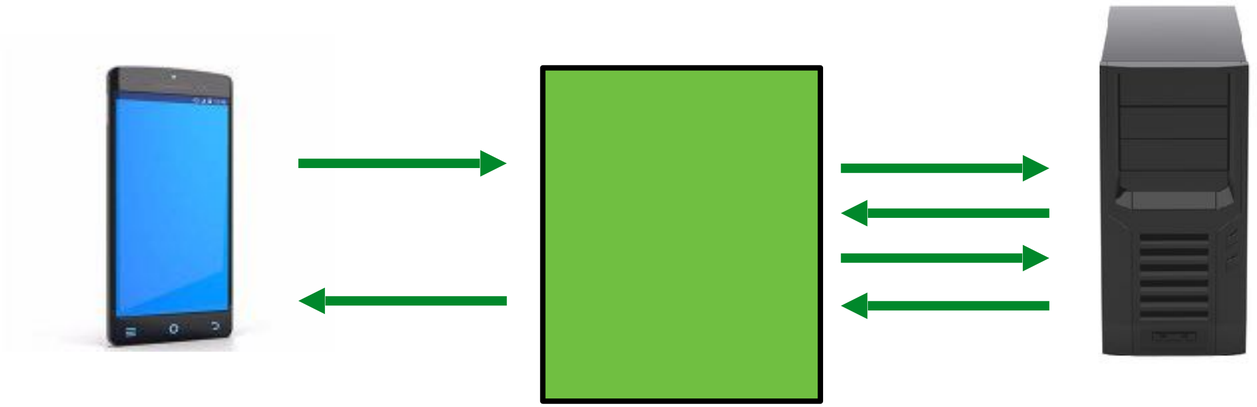
The depth of Batcher's sort is $O(\log^2 n)$: in a parallel computation model, only need $O(\log^2 n)$ steps.

→ 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



Generalizing

So far…

Traditional efficient sorting algorithms were not oblivious.

→ created new efficient oblivious sorting algorithm.

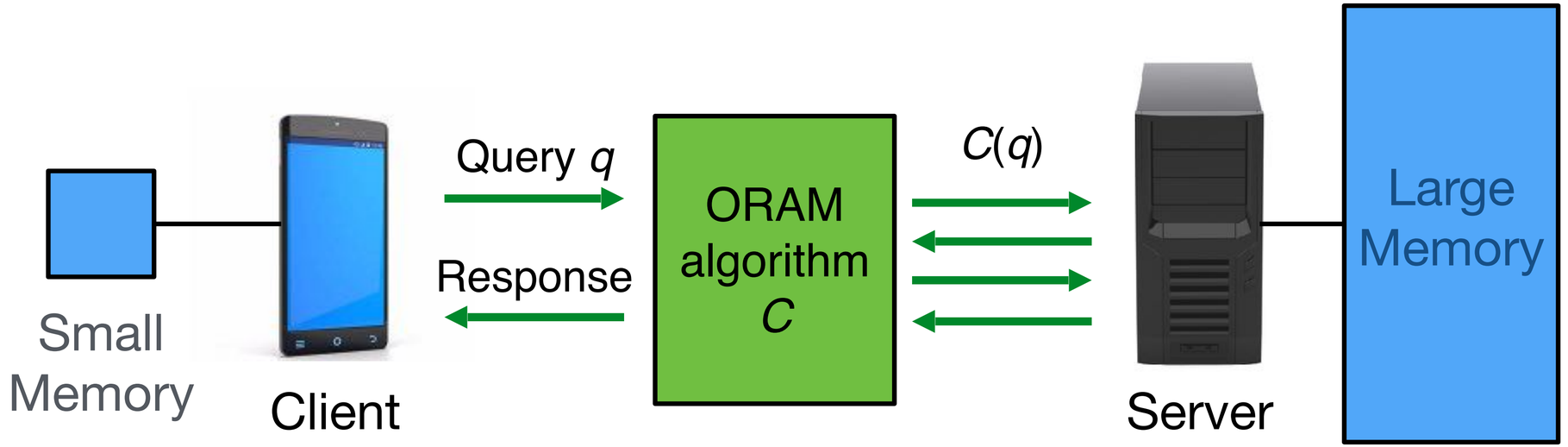
Can we do this generically?

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

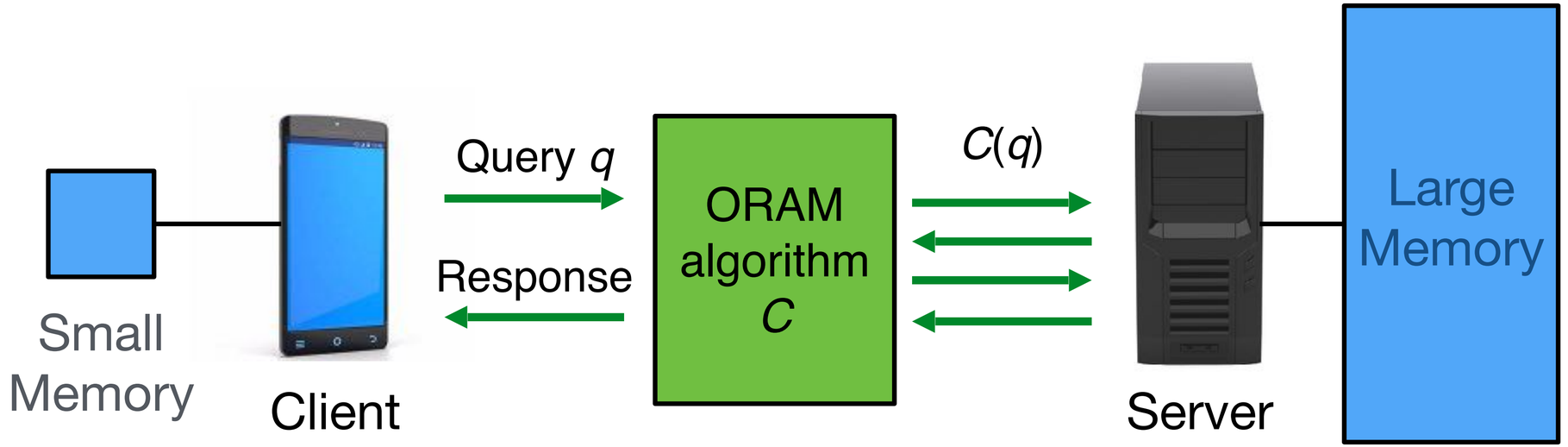


Client wants to do queries $q_1, q_2, \ldots, q_n$.

Each q_i is either:

- read(a): read data block at address a;
- write(a,d): write data block d at address a.

Reminder: Oblivious RAM

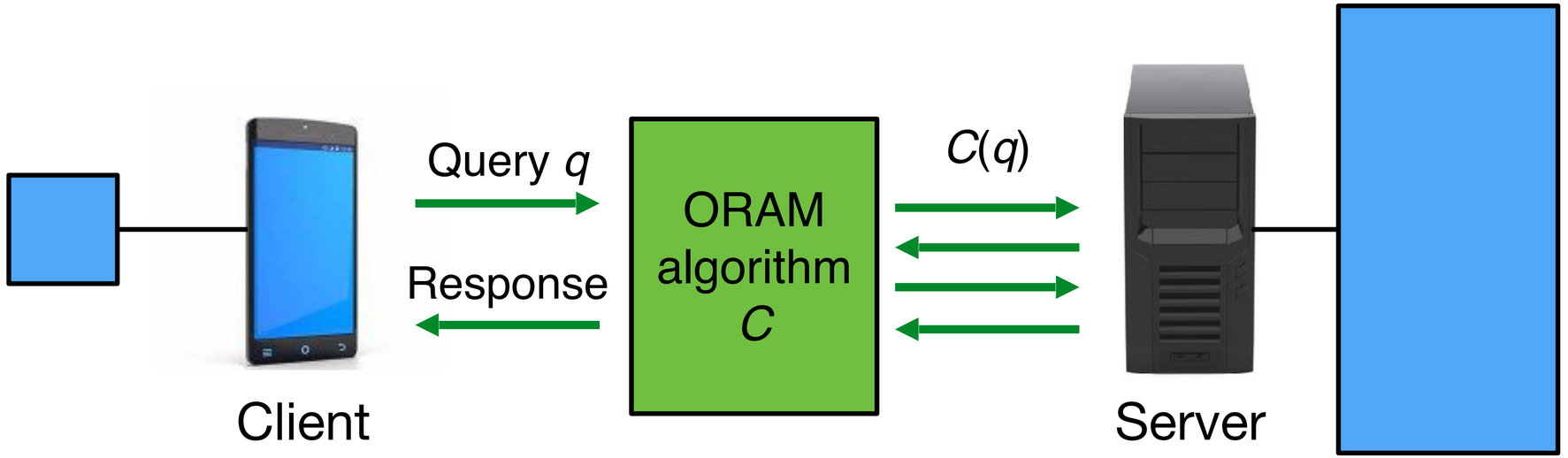


ORAM algorithm C (or ORAM "compiler"): transforms each query q by the client into one or several read/write queries $C(q)$ to server.

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



Trivial ORAM: read and re-encrypt **every** item in server memory.

Security: trivial.

Efficiency: every client query costs O(n) real accesses
→ overhead is O(n).

A non-trivial ORAM must have:

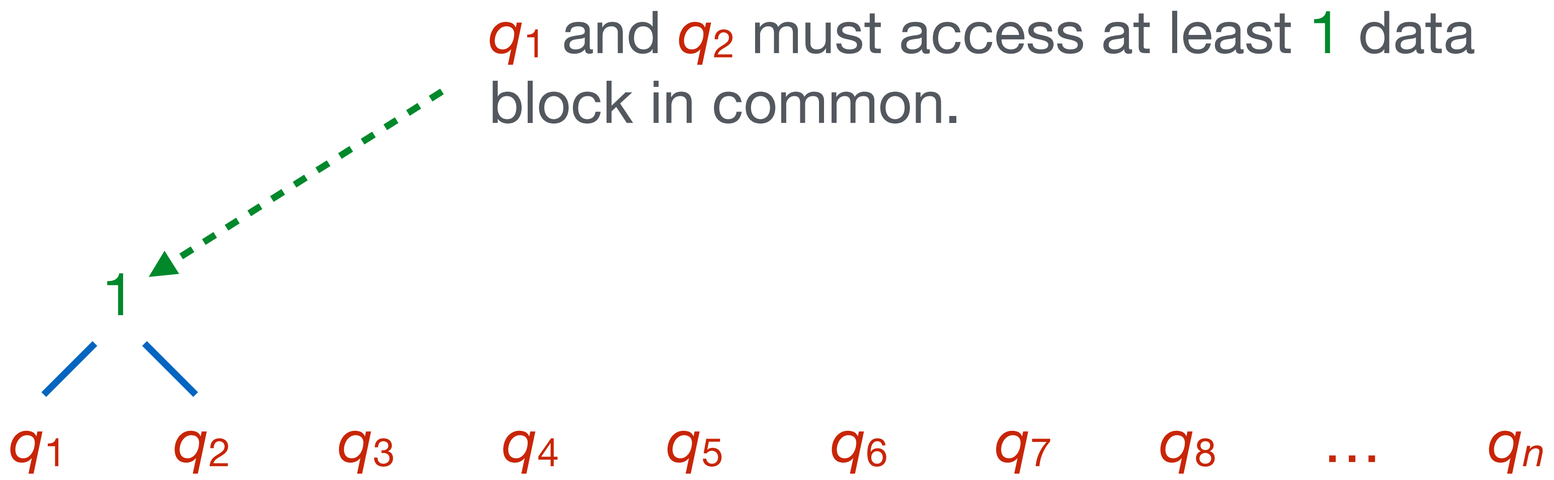
- Client storage o(n).
- Query overhead o(n).

Some observations

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

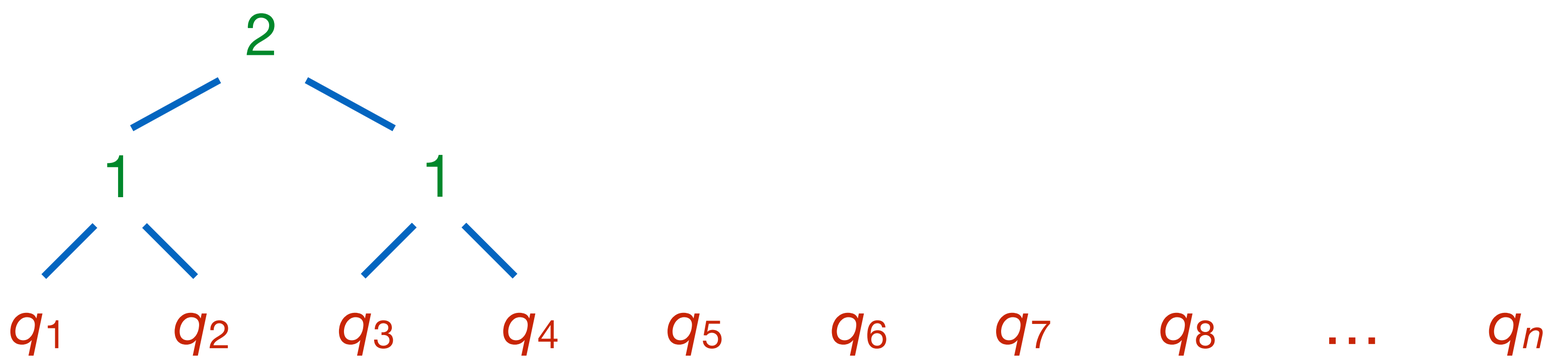
Assume the client does not store any memory block.

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



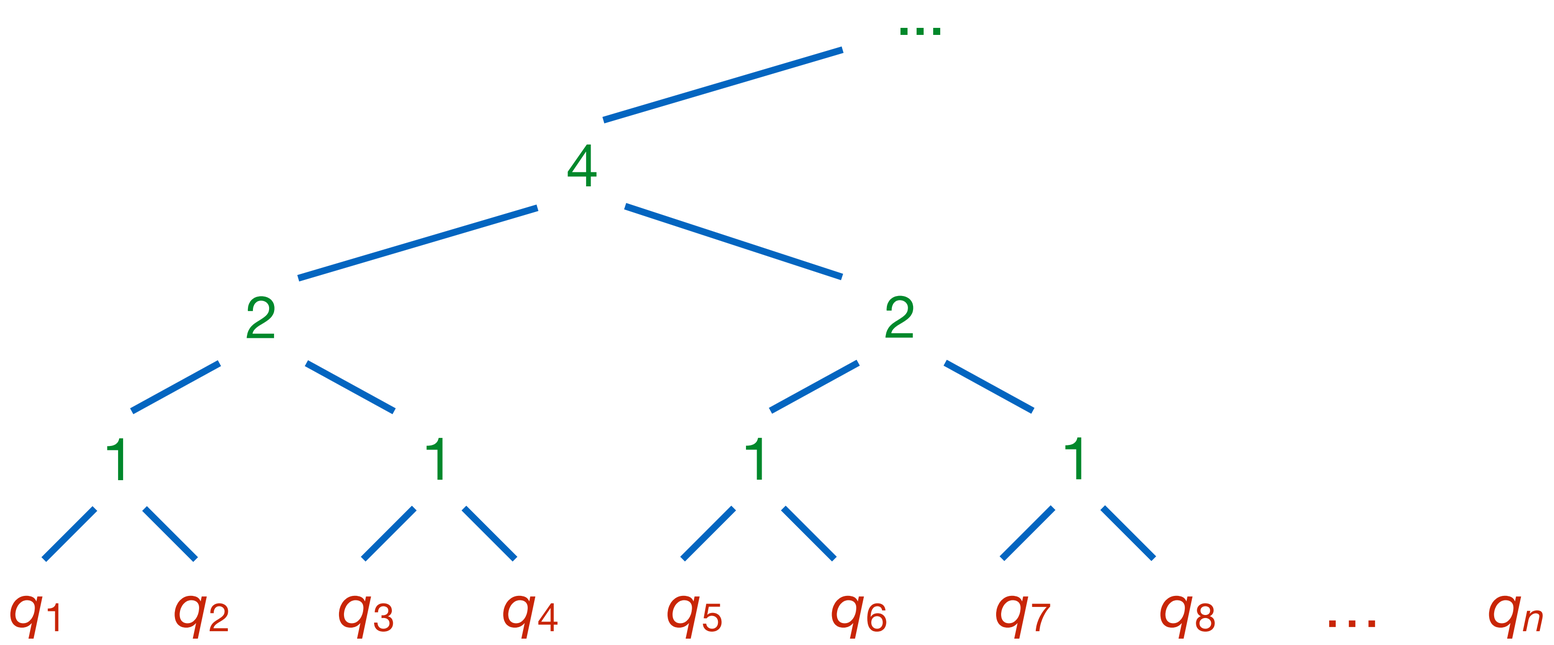
Some observations

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



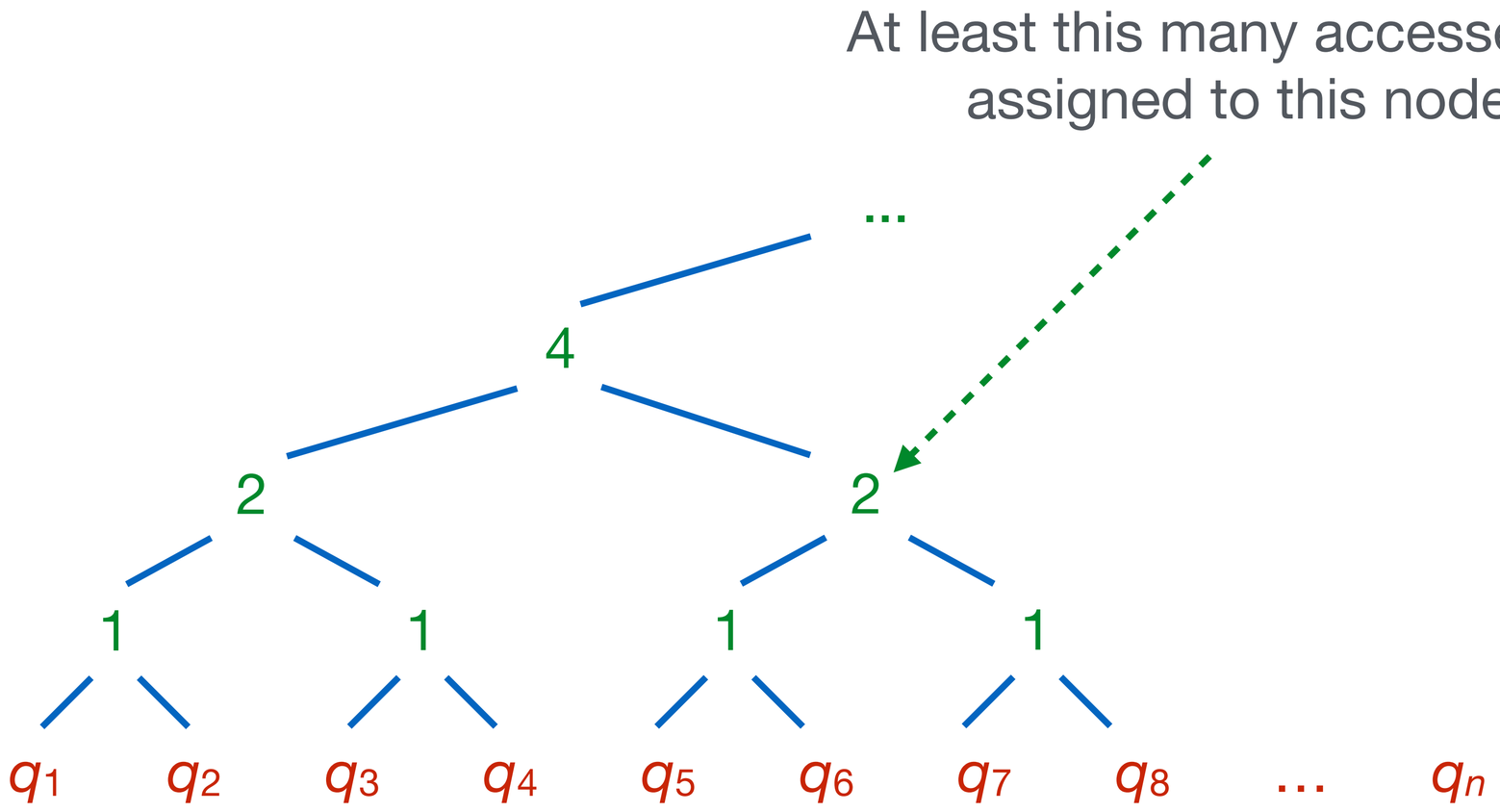
Some observations

etc...



Some observations

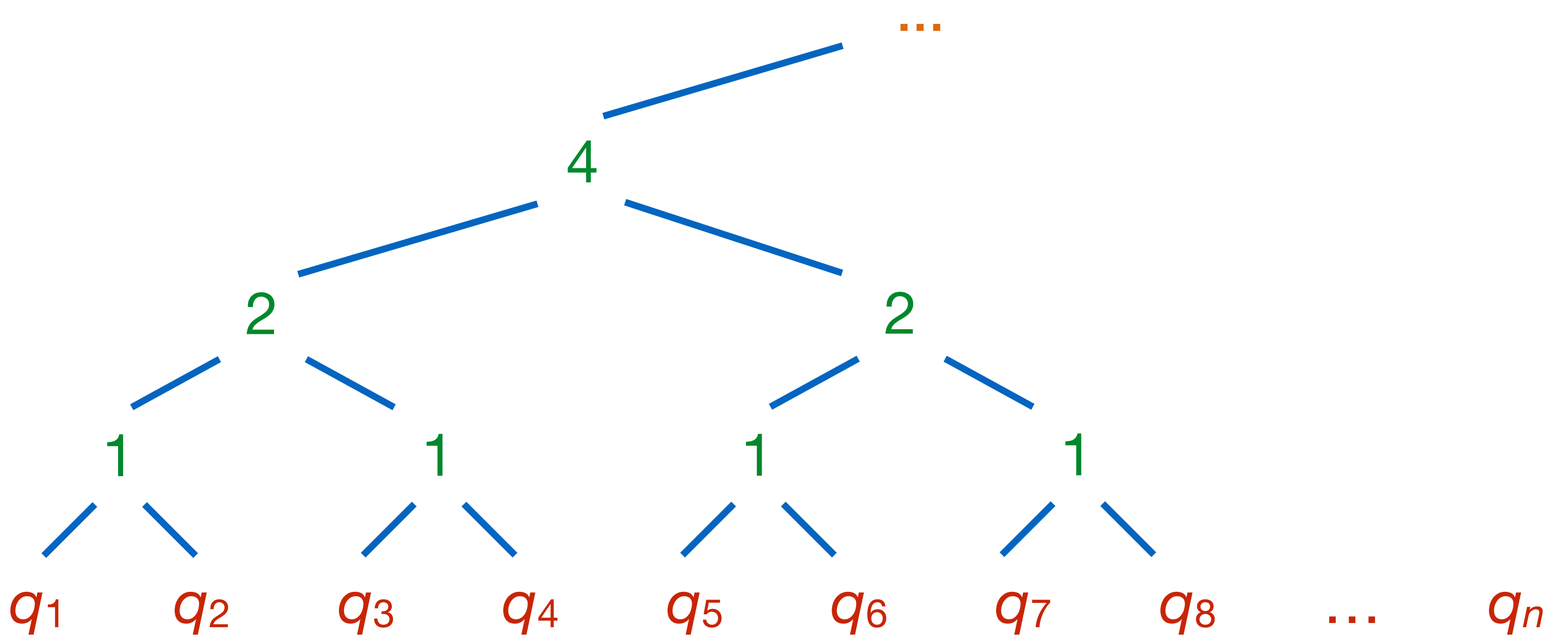
For each memory access done by q_j, let's "assign" that access to the node $q_i \wedge q_j$, where q_i is the last time the same address was accessed ($i < j$).



Some observations

For each memory access done by q_j, let's "assign" that access to the node $q_i \wedge q_j$, 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.

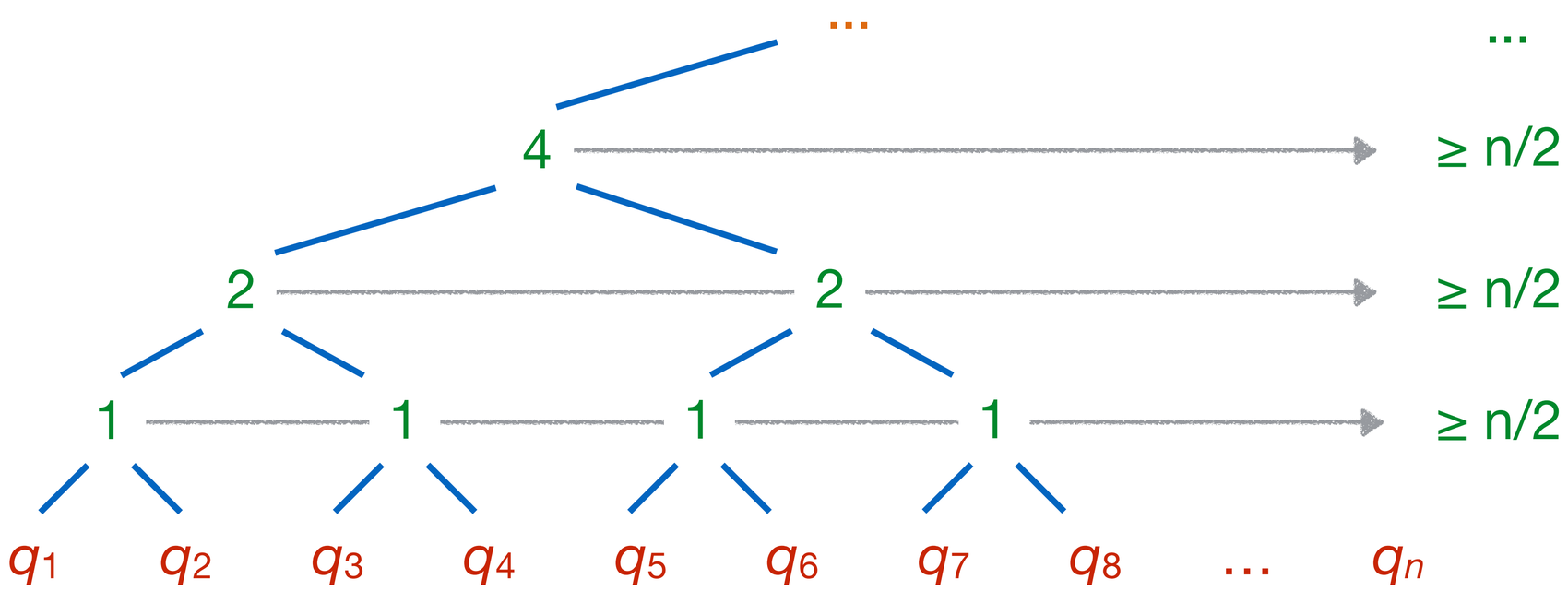


Some observations

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

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

For ***n*** client queries, ORAM will need to do $\Omega(\boldsymbol{n \log n})$ 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 = 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.

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

$$\Rightarrow \quad \boxed{f(q) = q \cdot \Omega(\log n)}$$

Roadmap

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

Here, n = max memory size = max number of items (*address*,*data*).

Family of constructions	Overhead	Feature
1. Square-root ORAM	$\tilde{O}(n^{1/2})$	Simple
2. Hierarchical ORAM	O(polylog n)	Best in theory
3. Tree ORAM	O(polylog n)	Best in practice

polylog(x) = poly(log(x)) = O(log^c(x)) for some constant c.

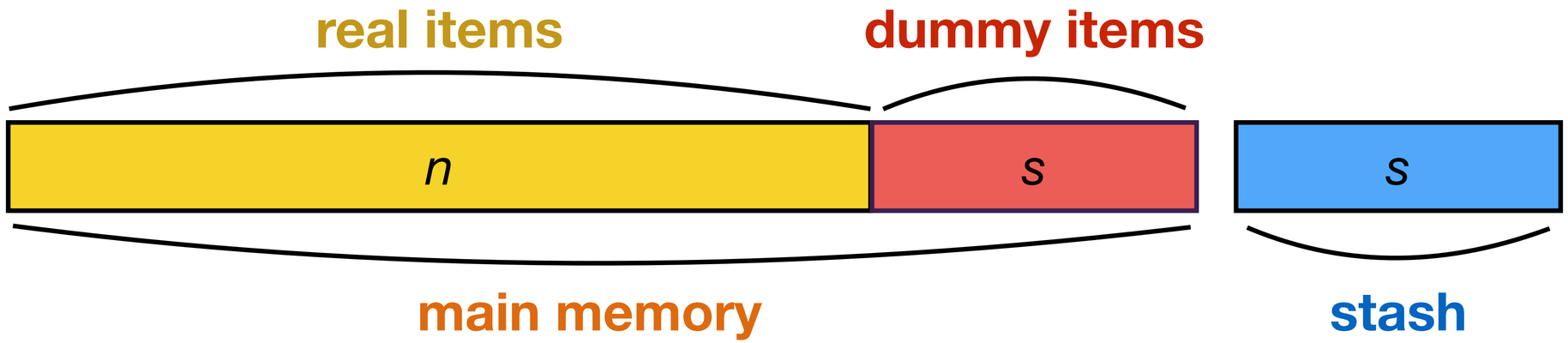
Other efficiency metrics: client memory size, number of roundtrips in $C(q)$, time complexity of C…

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:

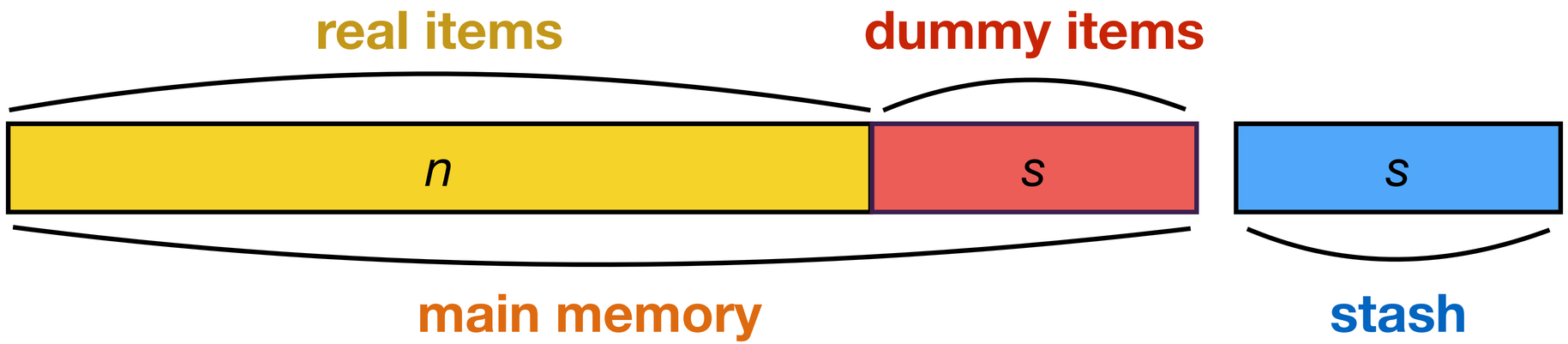


The **main memory** stores $n+s$ items:

- **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.

Setup



1. Client chooses permutation π over [1,*n*+*s*].

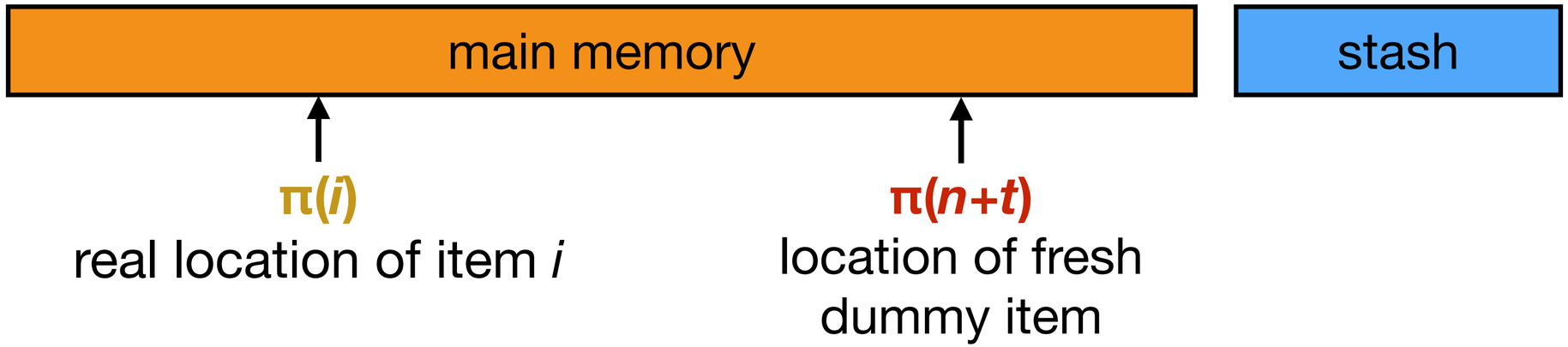
Item *i* will be stored at location π*(i)* in the main memory.

2. Client encrypts everything, and sends to server.

Server view:

Remark: we are assimilating client with ORAM algorithm.

Lookup



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 π(*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 π(*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 π'.
- Moves item i to location $\pi'(i)$ in main memory.
- Empties stash.

→ equivalent to fresh setup with π'.

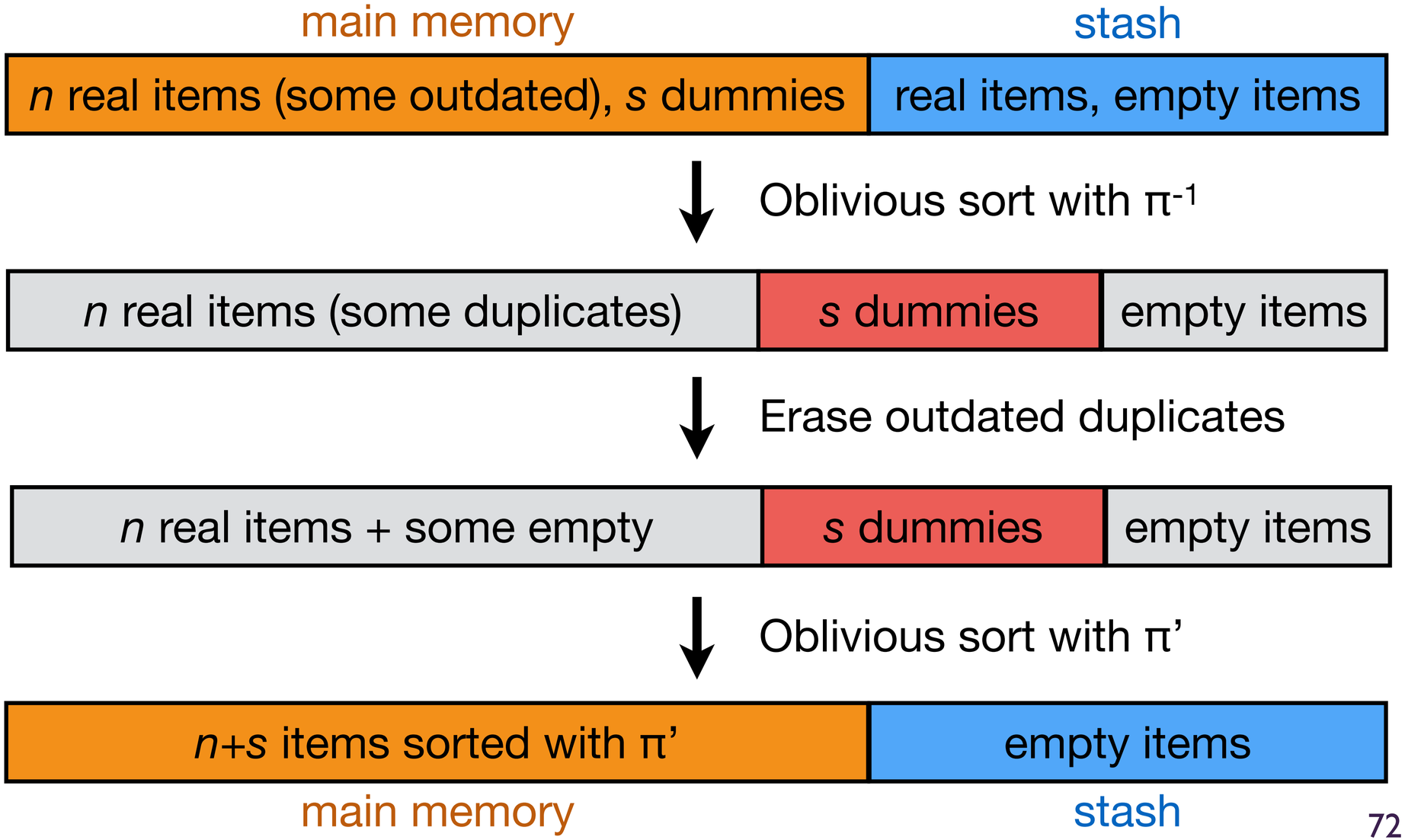
→ can do s iterations again…

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

Oblivious sorting!

Refresh via oblivious sort

Server memory after *s* lookups…



Security

Setup: server sees:

Lookup: server sees:

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 π.

Efficiency

Overhead.

Lookup costs O(s).

Refresh costs O(n 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:

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

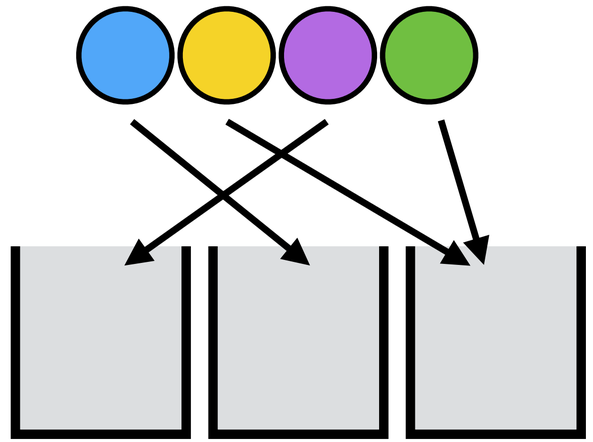
Server memory: O(n).

Client memory: O(1).

Need encryption key + key for pseudo-random π + few items during operations.

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

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:

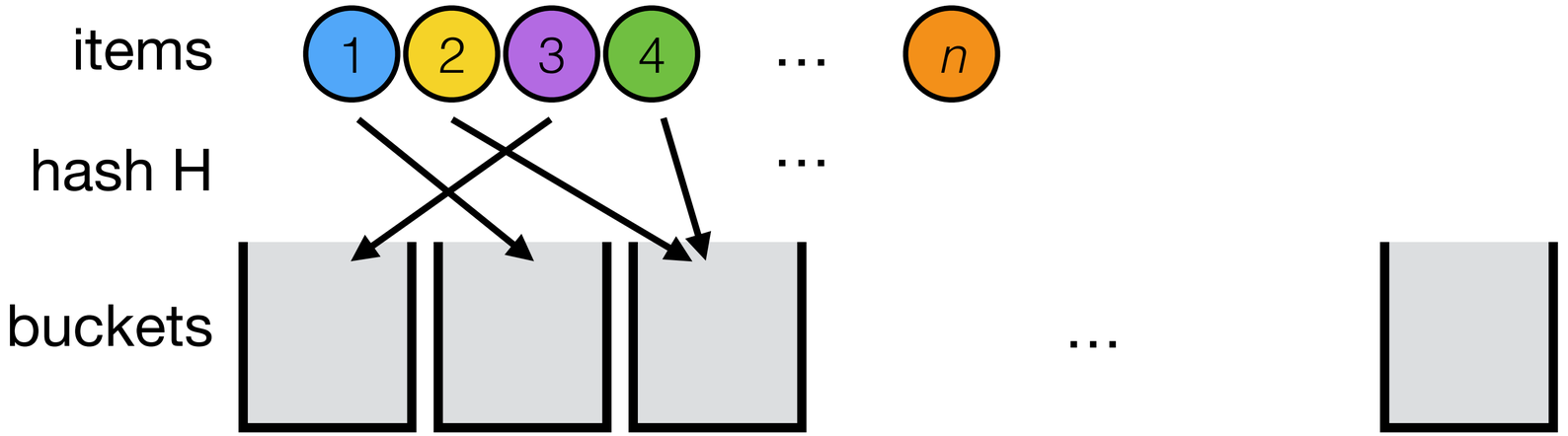
- First ORAM with polylog overhead.
- **Basis for the recent construction of optimal ORAM with overhead $O(\log n)$.**

Open problem for 20+ years, solved by Asharov et al. ’18, based on Patel et al. ’18.

Hashing

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

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



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

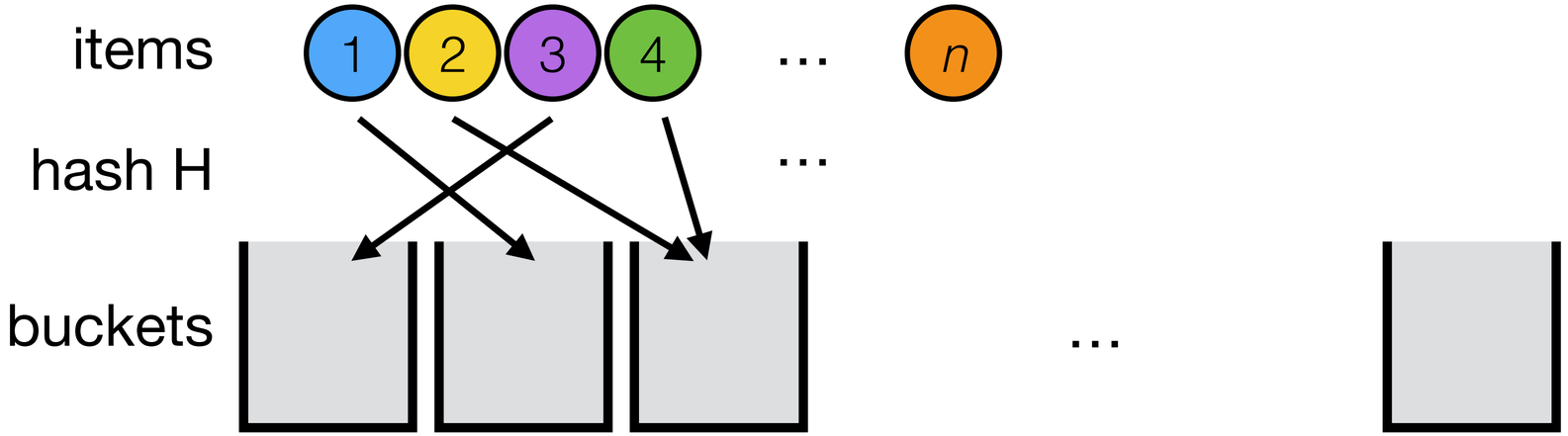
Proof: Probability that given bucket receives more than k items is $\exp(-\Omega(k^2))$ by Chernoff bound. Union bound over all buckets:

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

Oblivious hashing

Want to do the assignment obliviously…

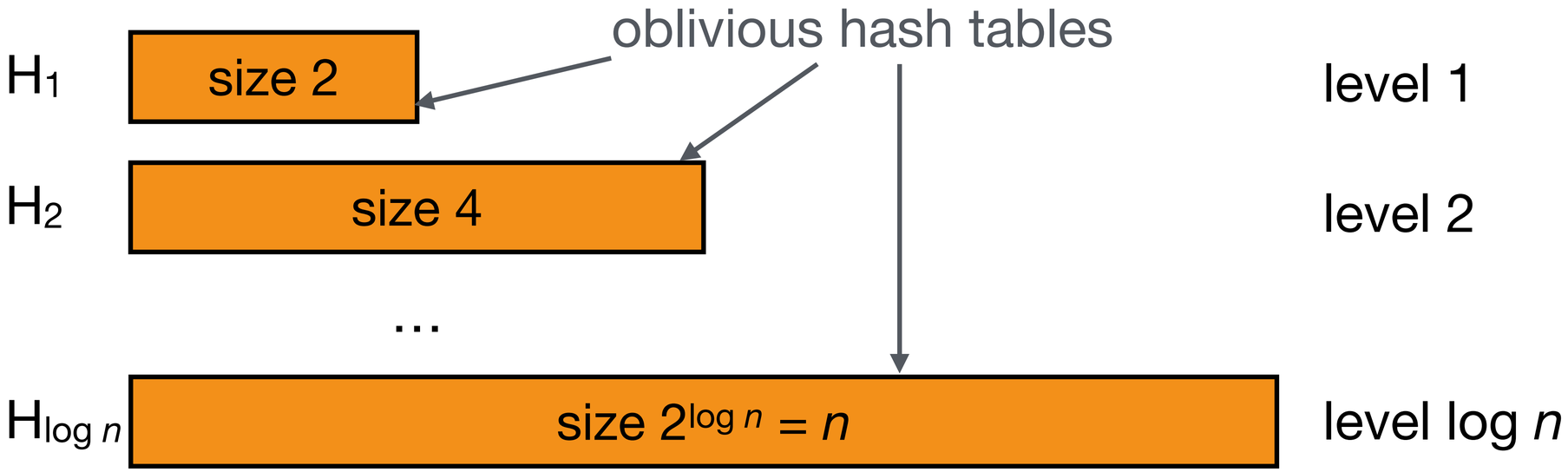
Suppose we have items + empty buckets all in server memory.



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.

Setup



Server memory arranged into log n levels.

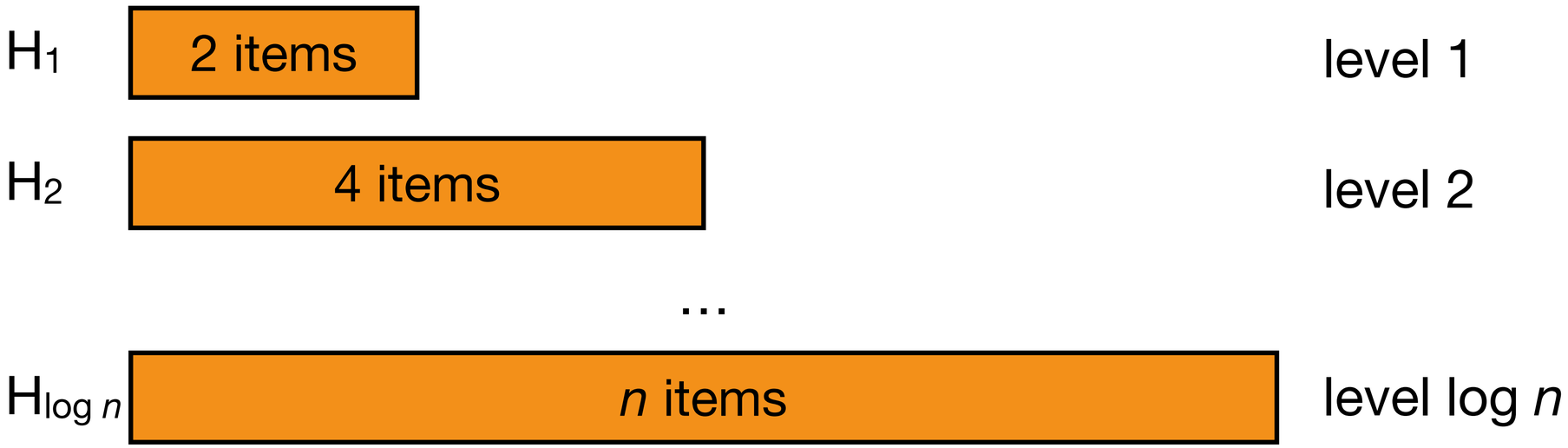
Each level k is an (oblivious) hash table for 2^k items.

At start:

All items are in last level.

Other levels contain dummies.

Lookup



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.

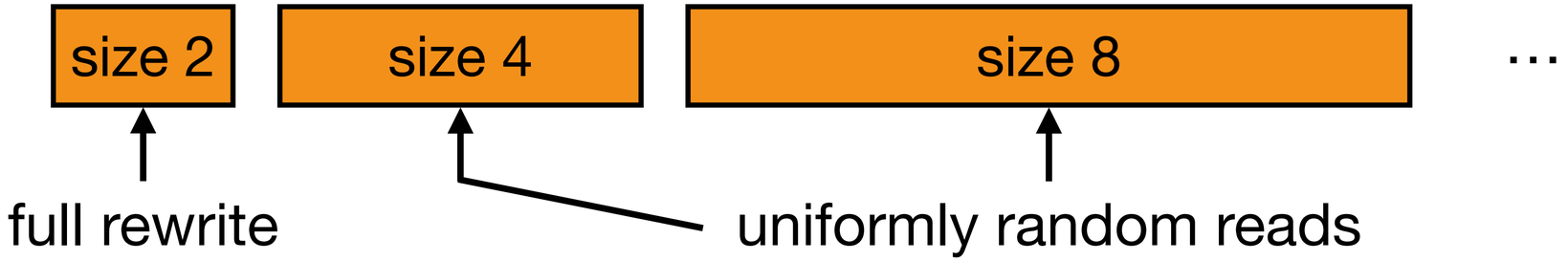
And empties its content every 2^k lookups.

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:

Lookup: server sees:



+ 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: $O(1)$.

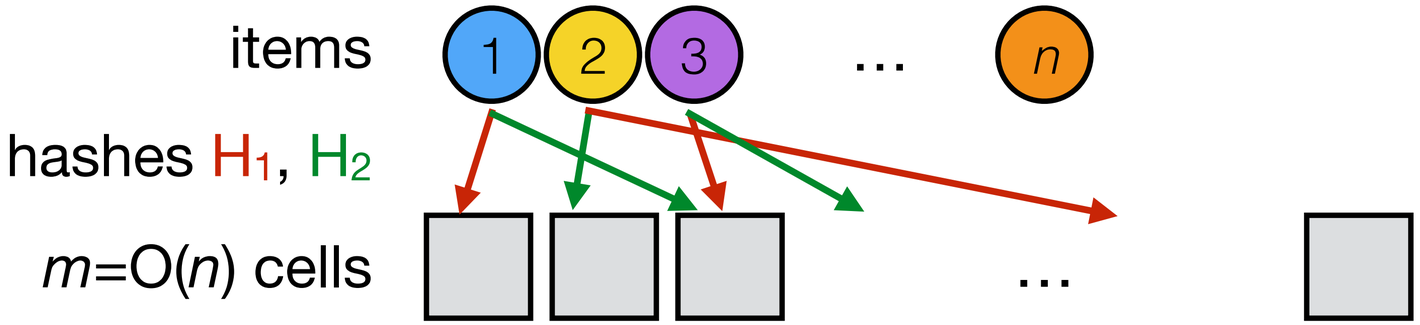
Server memory can be reduced to $O(n)$ using **cuckoo hashing**.

Cuckoo hashing

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

Initial design mainly motivated by real-time systems…

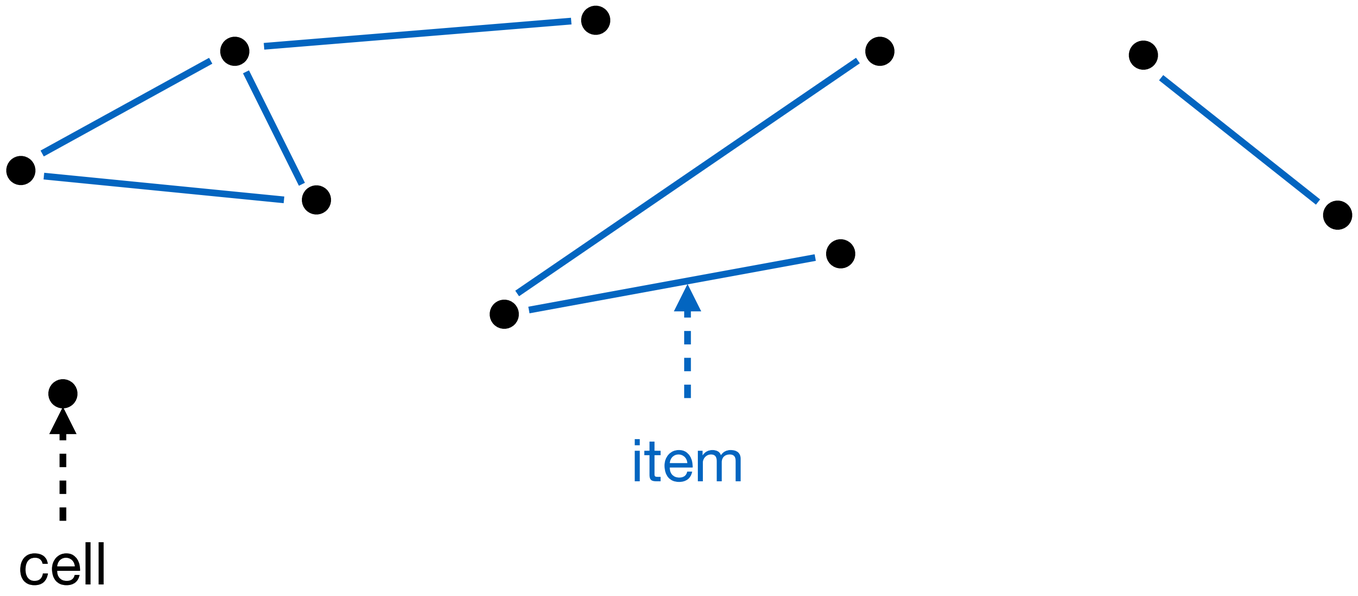
Idea:



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

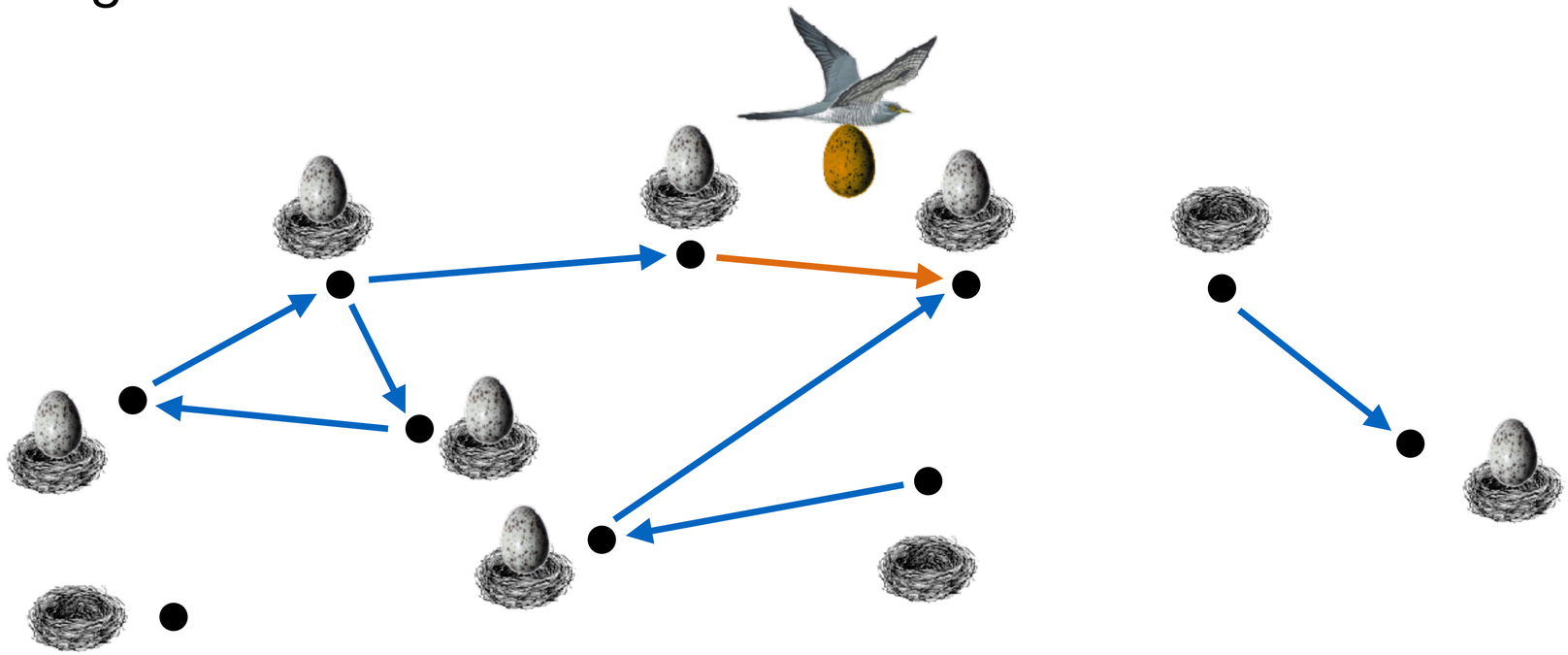
The cuckoo graph

Picture graph with cells = nodes, item *i* = edge $H_1(i)$ - $H_2(i)$.



The cuckoo graph

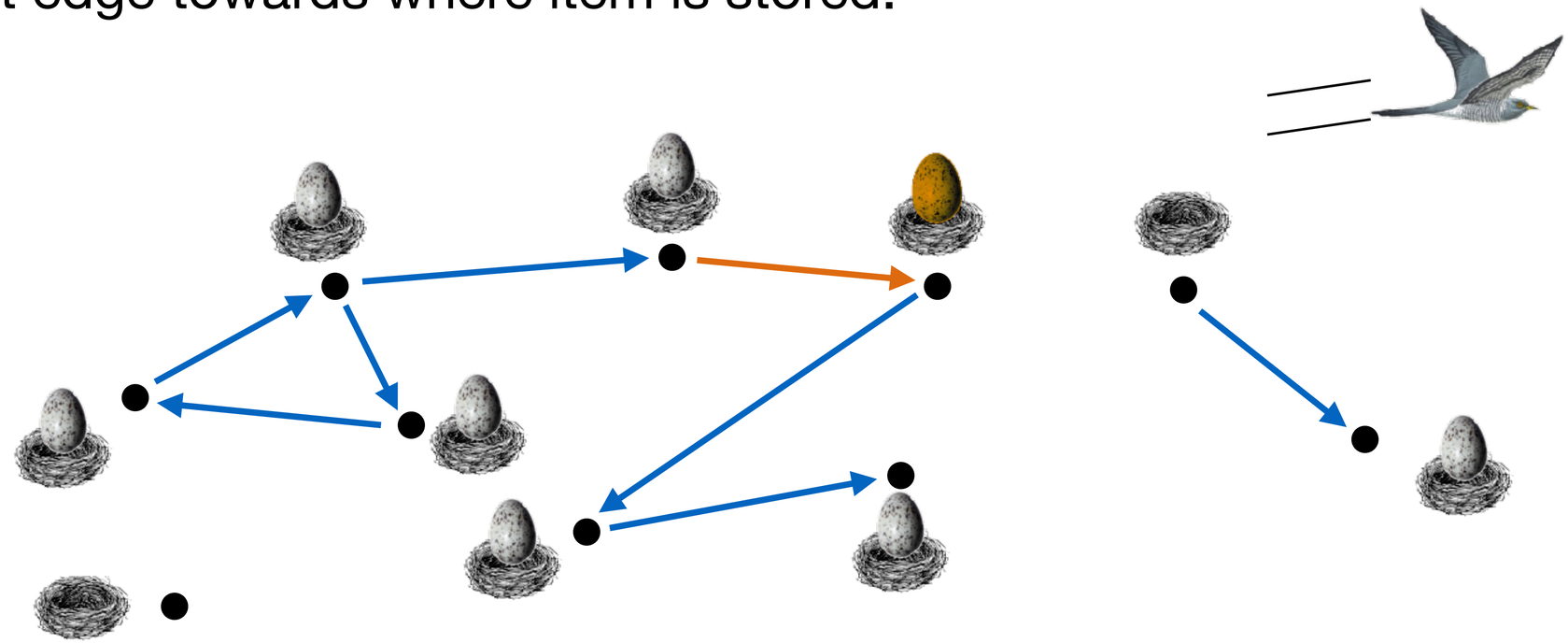
Picture graph with cells = nodes, item *i* = edge $H_1(i)$ - $H_2(i)$.
Orient edge towards where item is stored.



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.

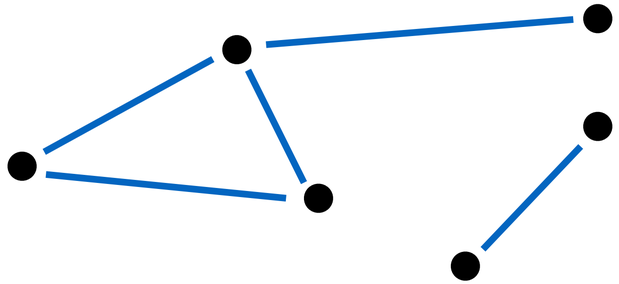
The cuckoo graph

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

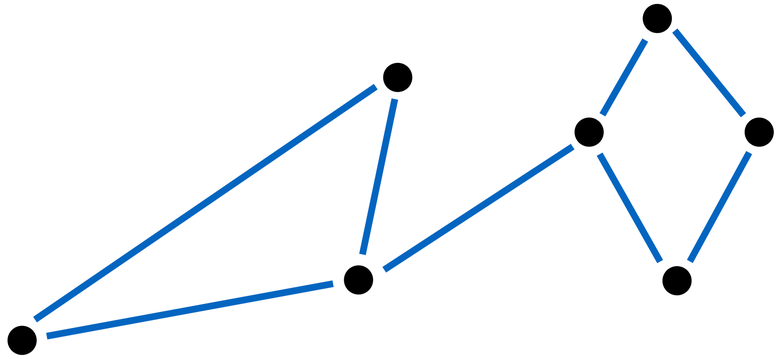


To insert item *i*: try cell $H_1(i)$. If occupied, move occupying item into its other possible cell. Repeat until unoccupied cell is reached.

Why does that work? (sketch)



possible



impossible

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

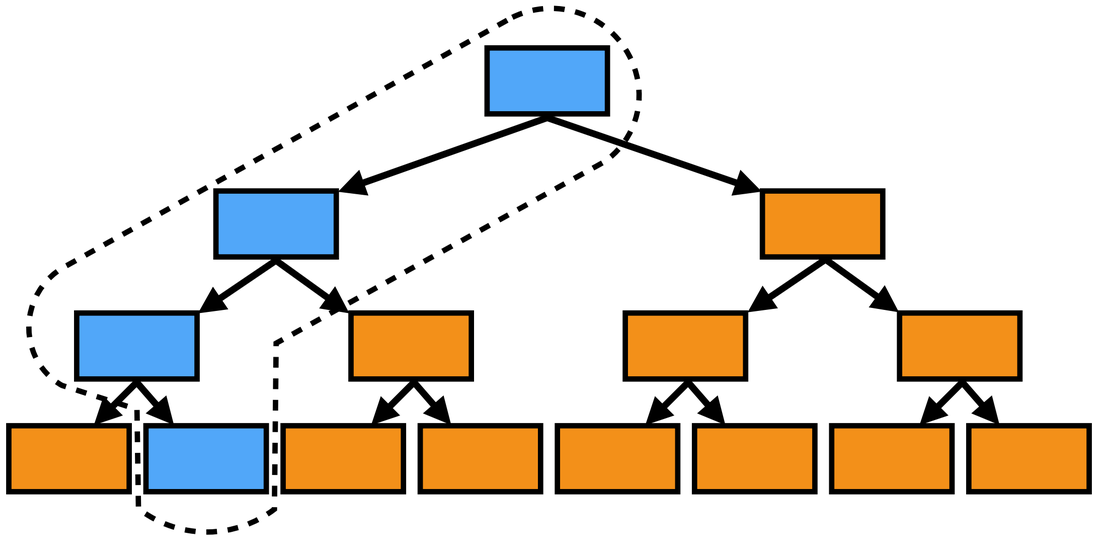
Moreover, with n edges and $m = (2+\varepsilon)n$ nodes…

- 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



Tree ORAM

Hierarchical ORAM family leads to recent optimal construction.
But huge constants. Never used in practice.

What is actually used:

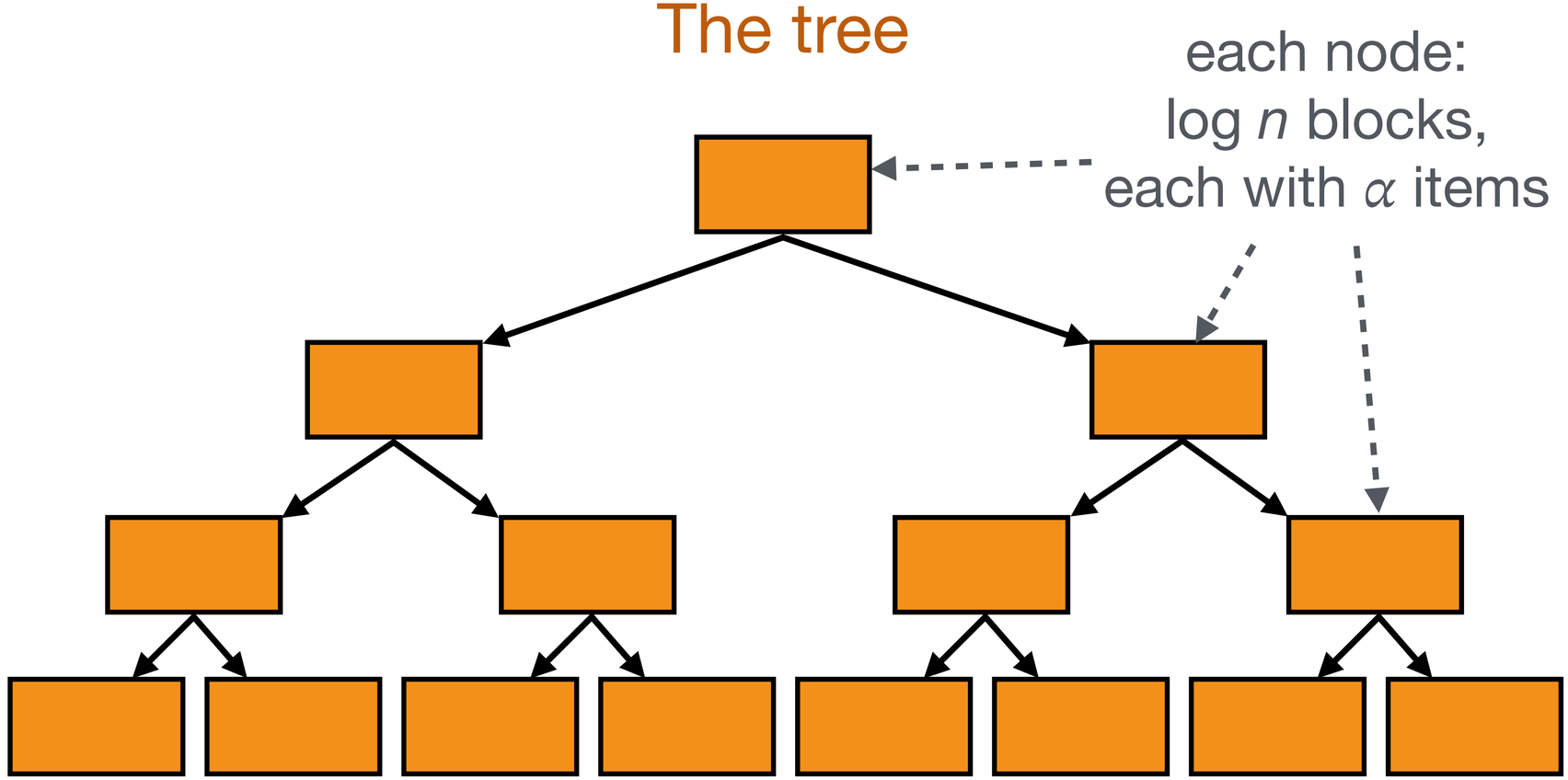
Tree ORAM *by Shi et al. ’11*

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.

The tree

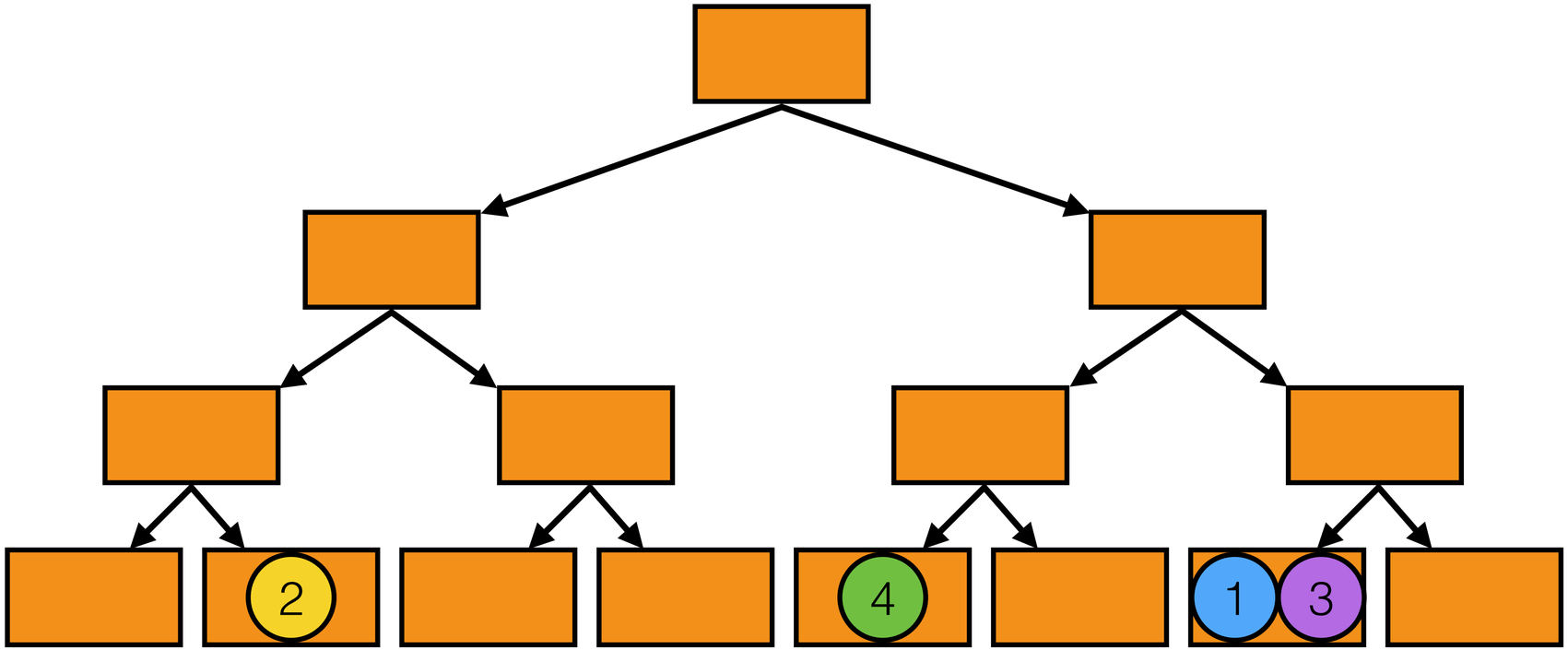


Server-side memory is a full binary tree with $\log(n/\alpha)$ levels.

Each node contains $\log n$ blocks.

Each block contains $\alpha = O(1)$ (possibly dummy) items.

Setup



Items are grouped into blocks of α items, item i into block $b = \lfloor i/\alpha \rfloor$.

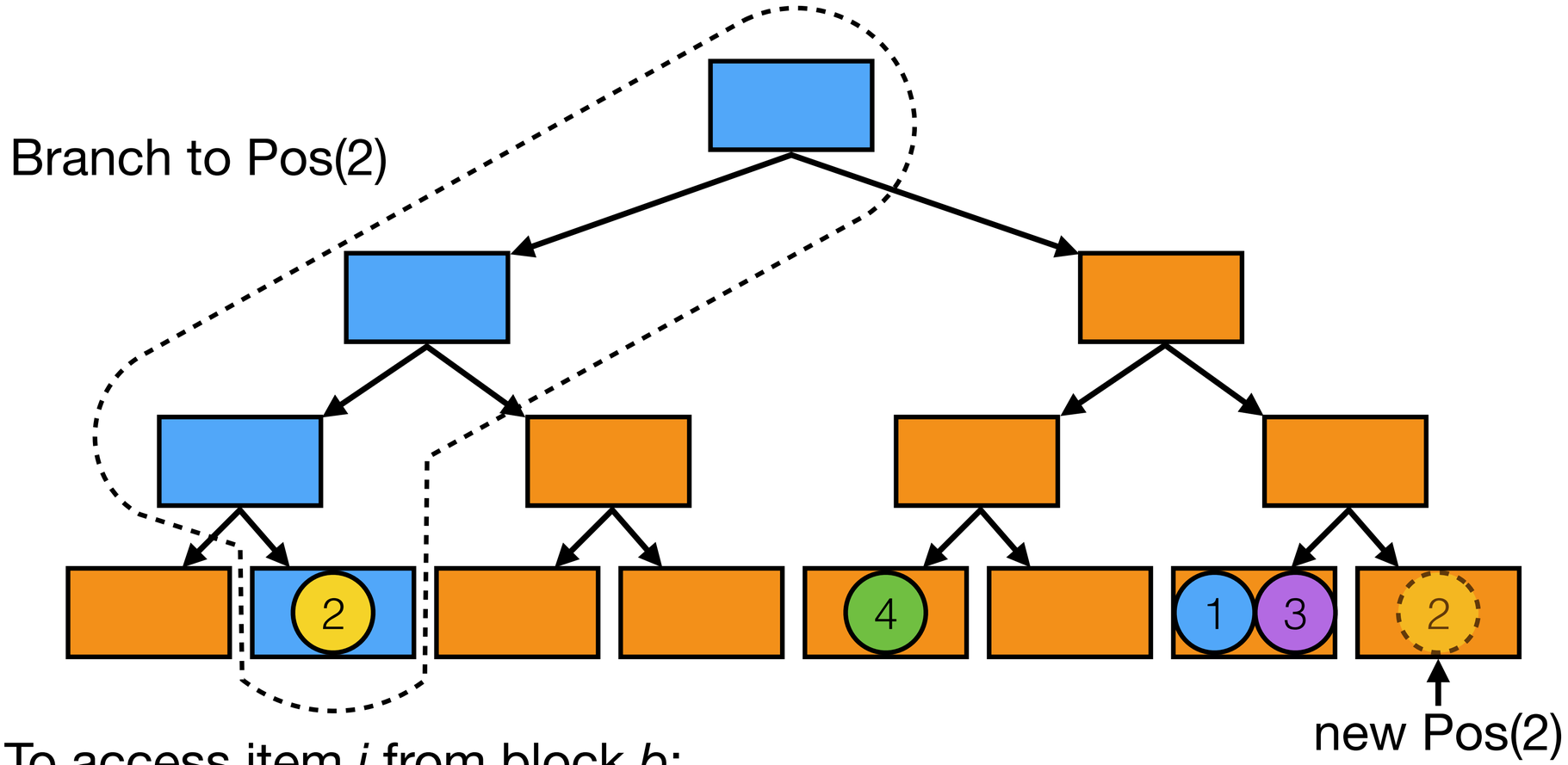
At start:

Each block b is stored in a uniformly random leaf Pos(b).

“Position map” Pos() is stored on the client.

Invariant: block b will always be stored on the branch to Pos(b).

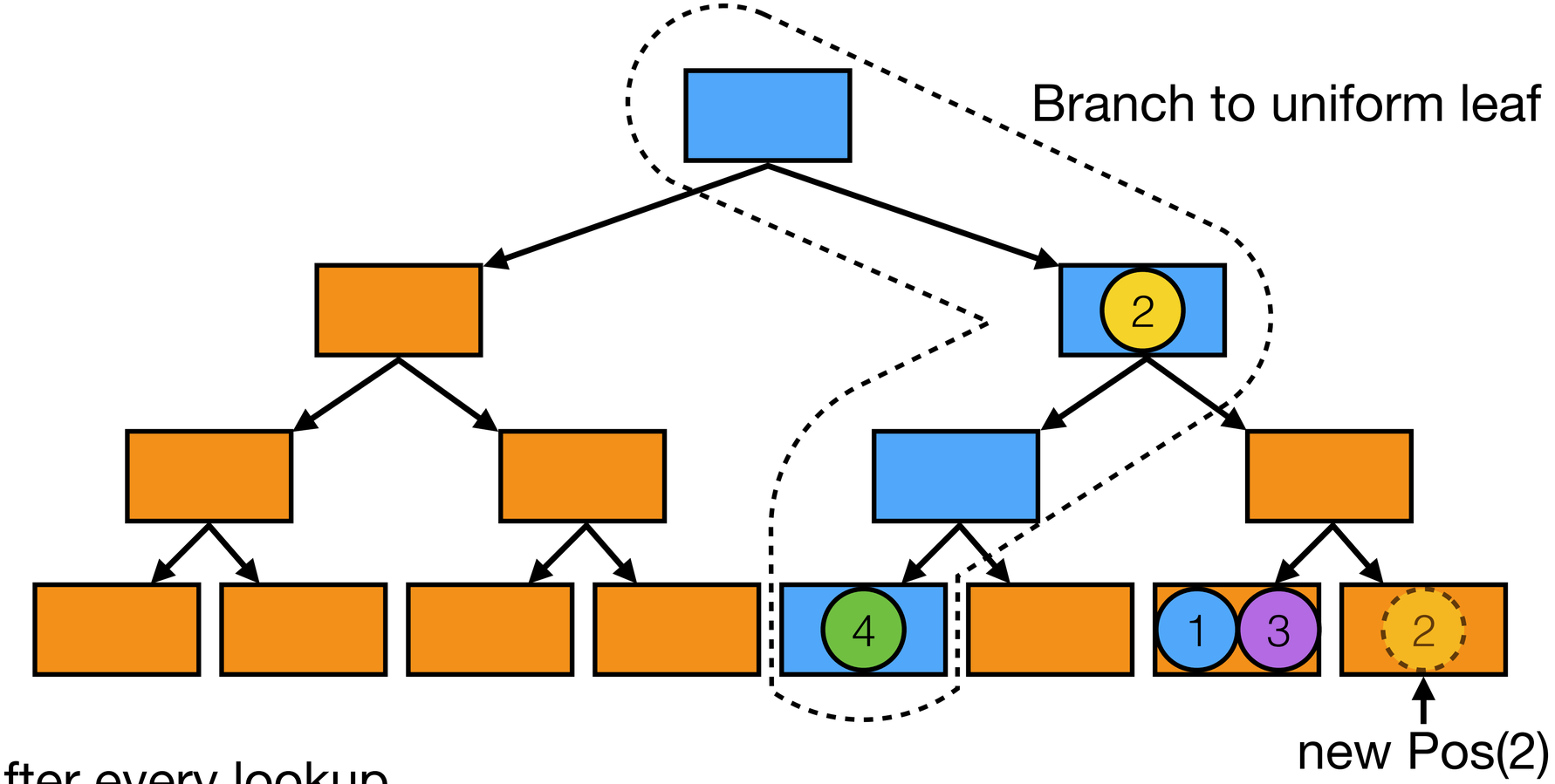
Lookup



To access item *i* from block *b*:

1. Read every node along branch to Pos(*b*). Remove *b* when found.
2. Update 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.

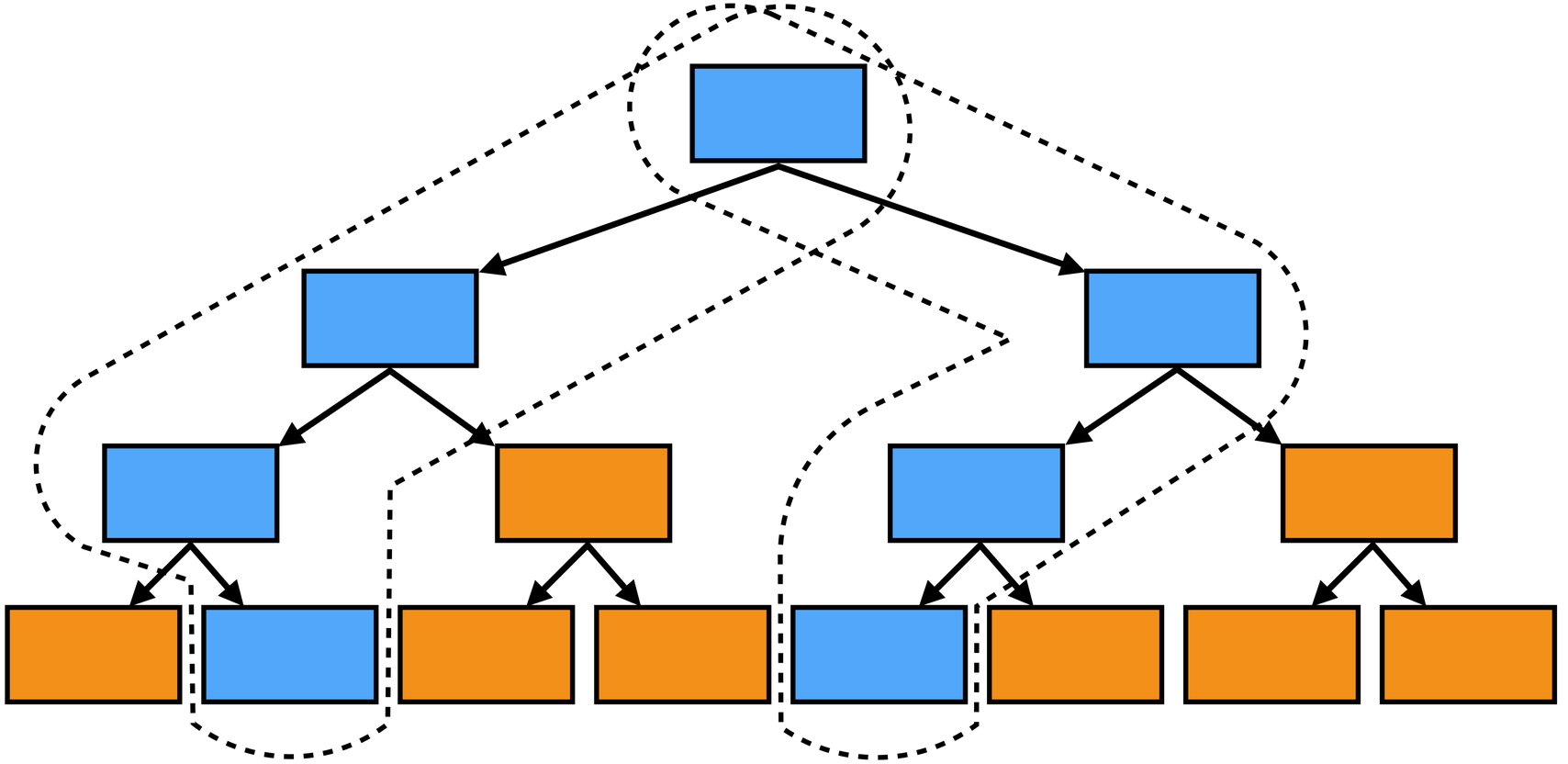
2.Push every block in the branch as far down as possible (preserving that block *b* must remain on branch to Pos(*b*)).

Security

Setup: server sees full binary tree of height log (n/α).

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 [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/α] → [1,n/α], 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 α 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, ecah of size O($\log^2 n$).

→ O($\log^3 n$) overhead.

Server memory: O(n log n).

Client memory: O(1).

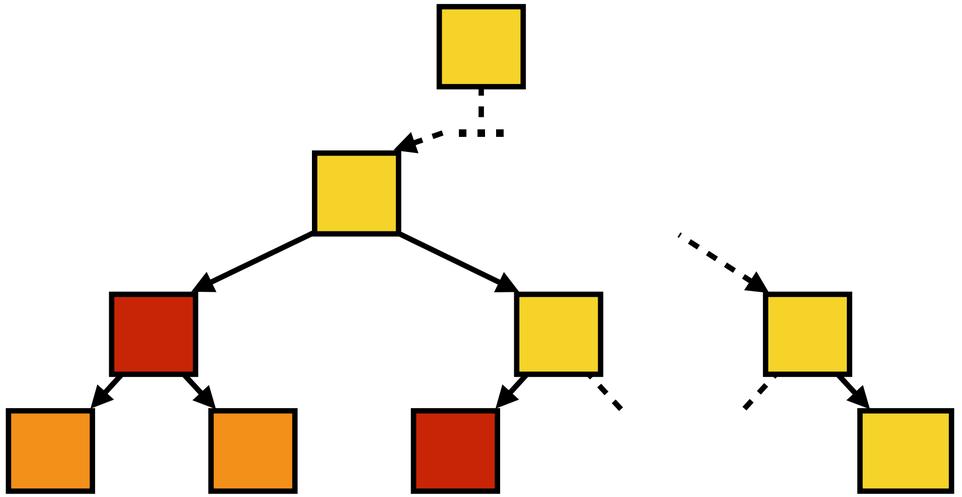
Variants

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

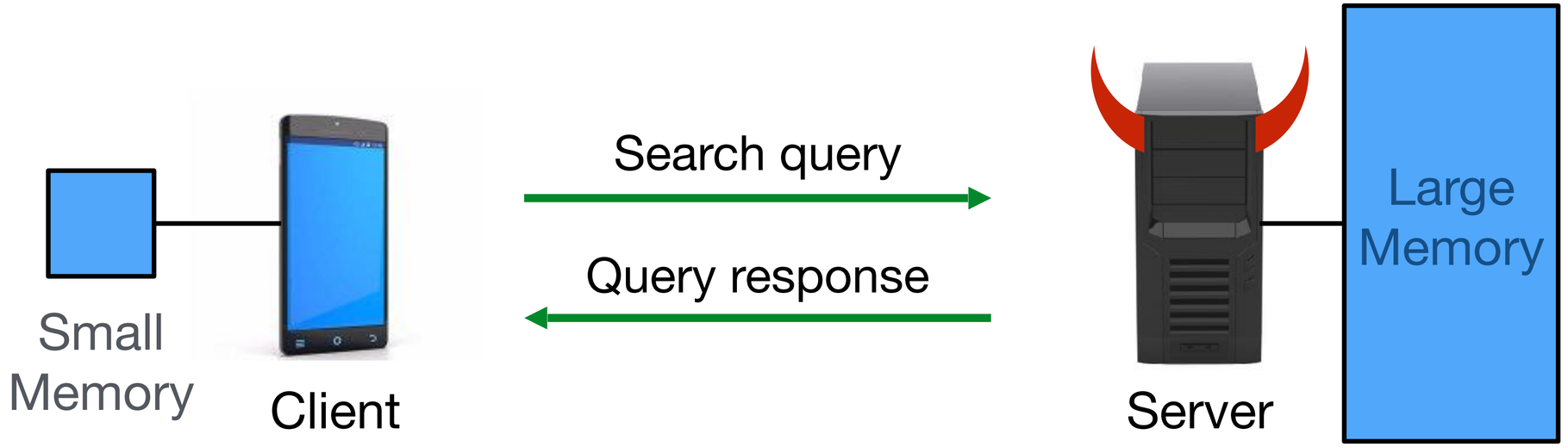
Path ORAM:

- 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 anwers.)

- Inefficient.

  (In certain cases, such as Enron email dataset or English Wikipedia, some studies suggests trivial ORAM would be most efficient.)

Idea of Searchable Encryption: allow *some* leakage. Gain in efficiency.

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.

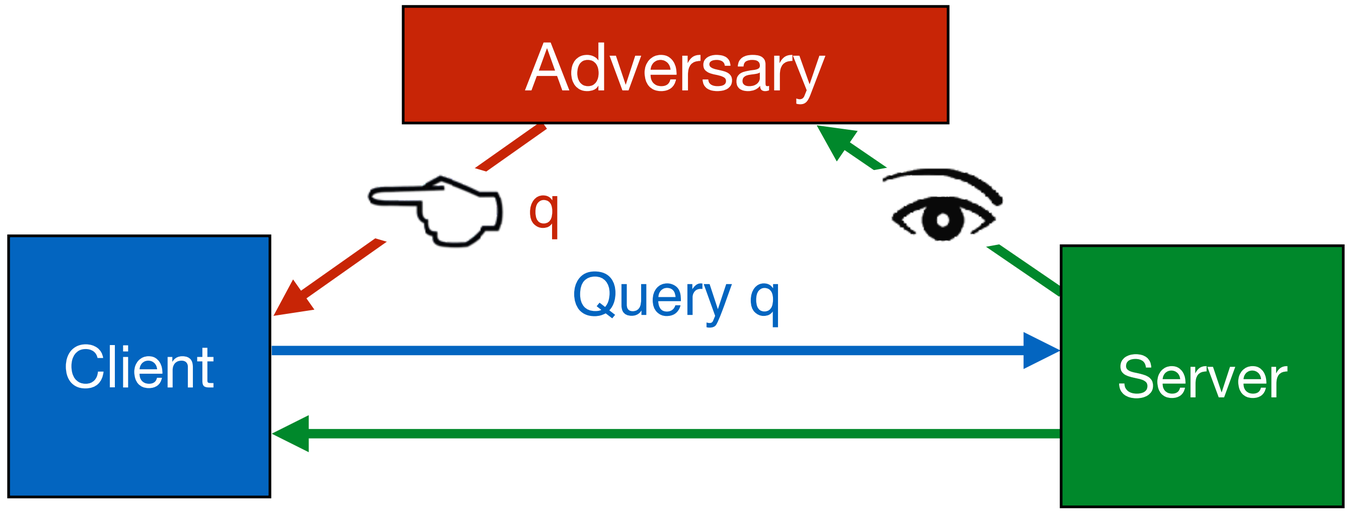
Searchable encryption security: accesses can be simulated by a simulator knowing only the output of a leakage function **L**.

- **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



Ideal world