Techniques in Cryptography and Cryptanalysis

Oblivious RAM

Brice Minaud

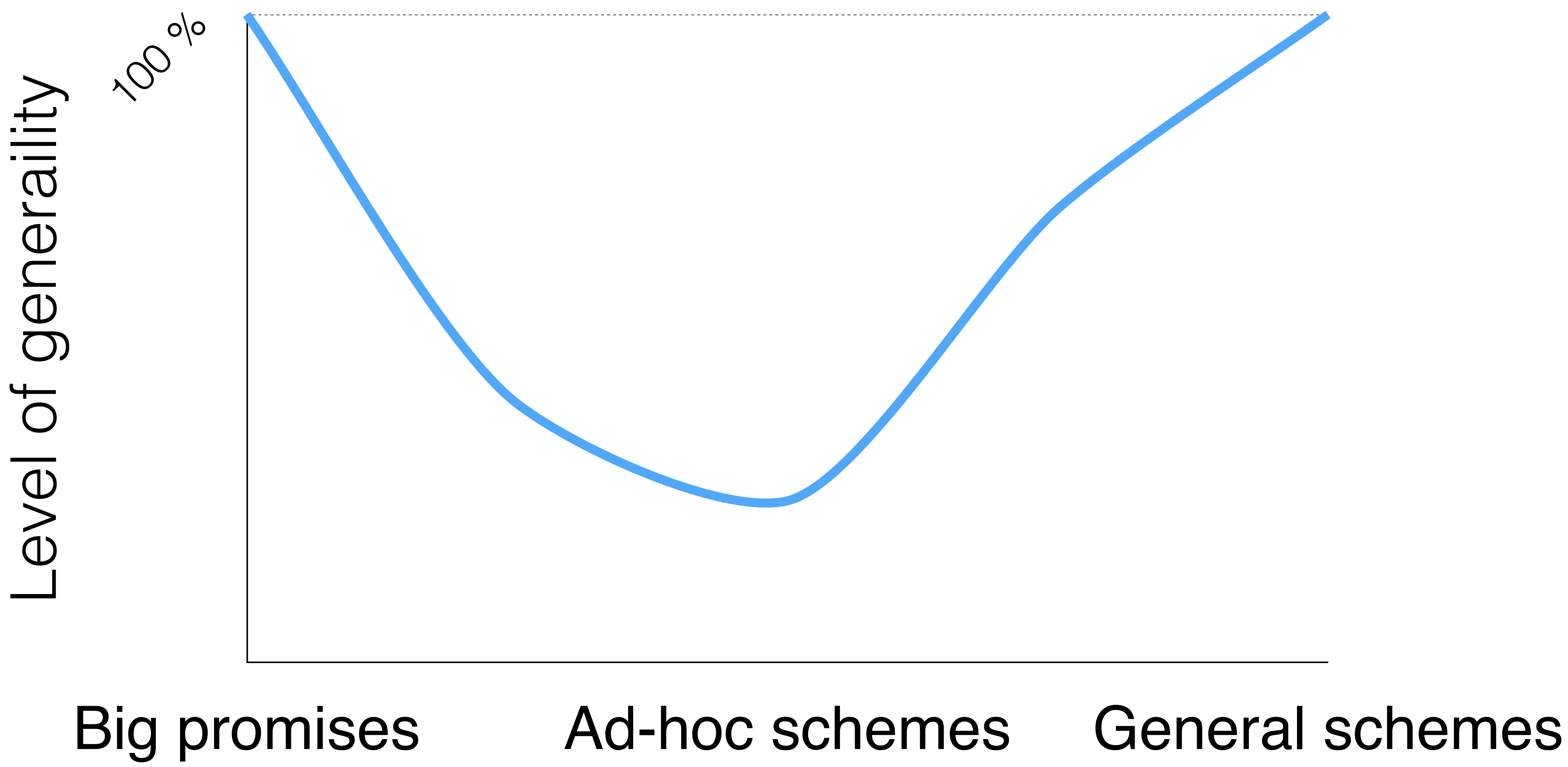
email: brice.minaud@ens.fr

MPRI, 2024-25

Meta information

- **Zero-knowledge proofs** ✔ done
- **Oblivious RAM** 📍 *we are here*
- **Fully Homomorphic Encryption**

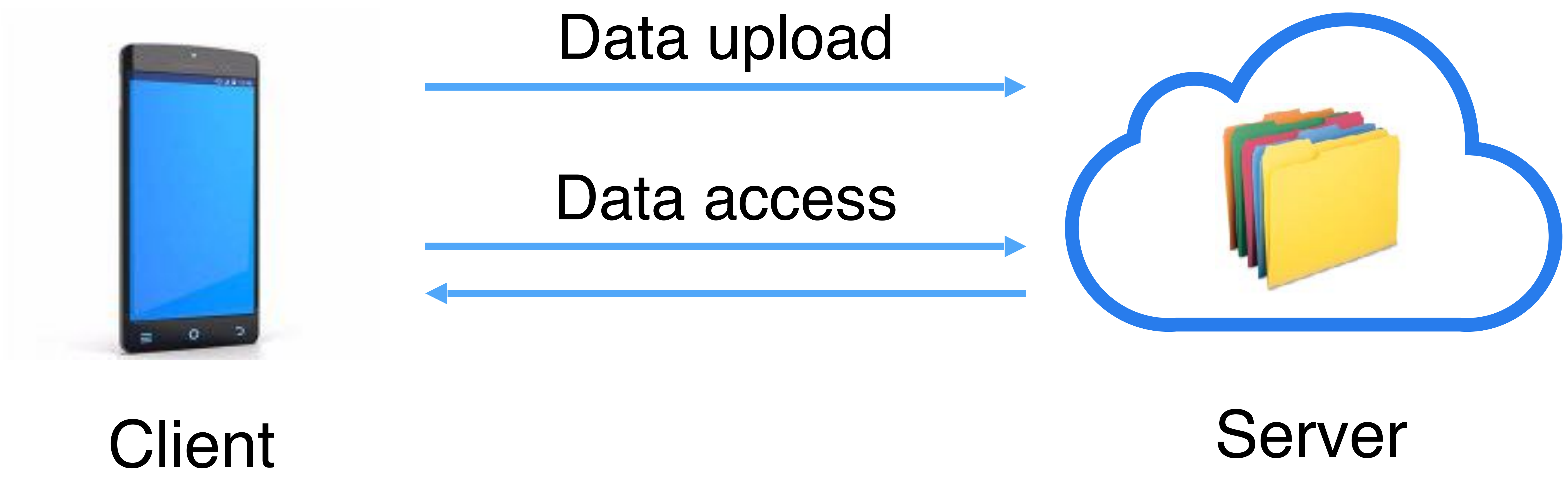
Roadmap



Course contents:

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

Outsourcing storage

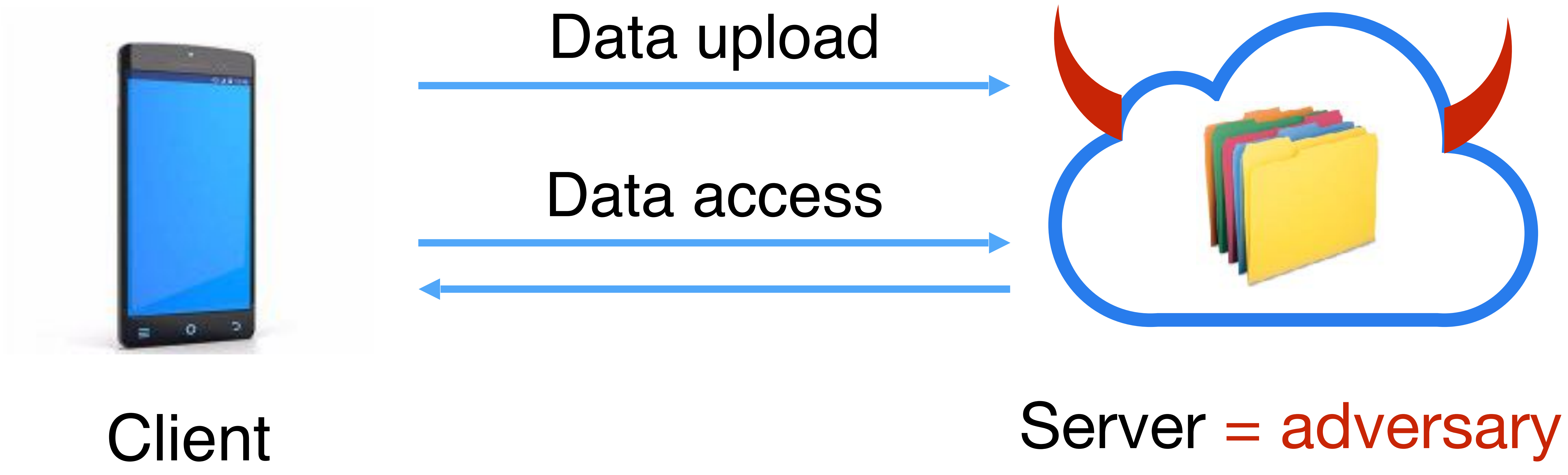


Scenario: Client outsources storage of sensitive data to Server.

Examples:

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

Outsourcing storage



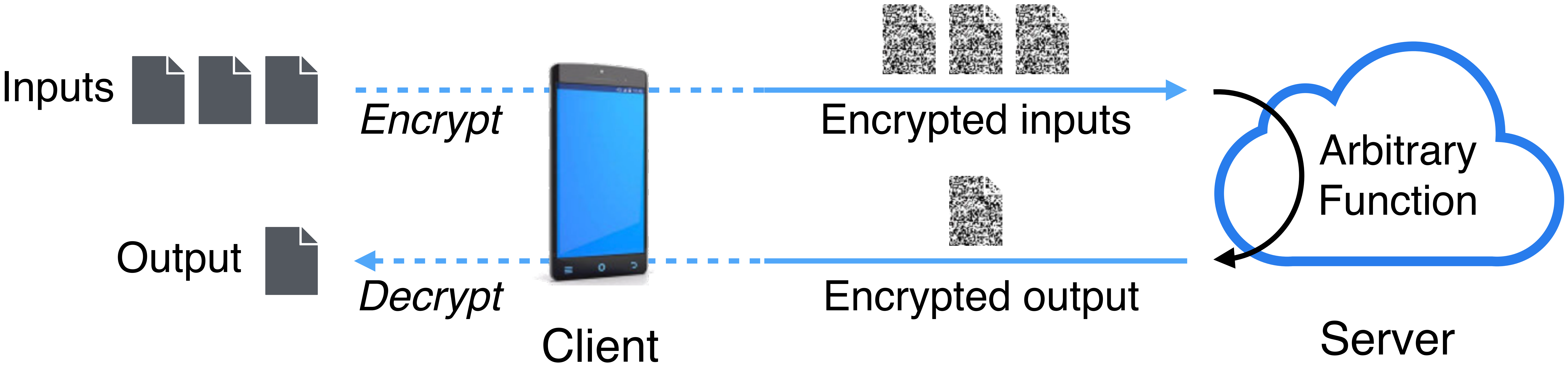
Scenario: Client outsources storage of sensitive data to Server.

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

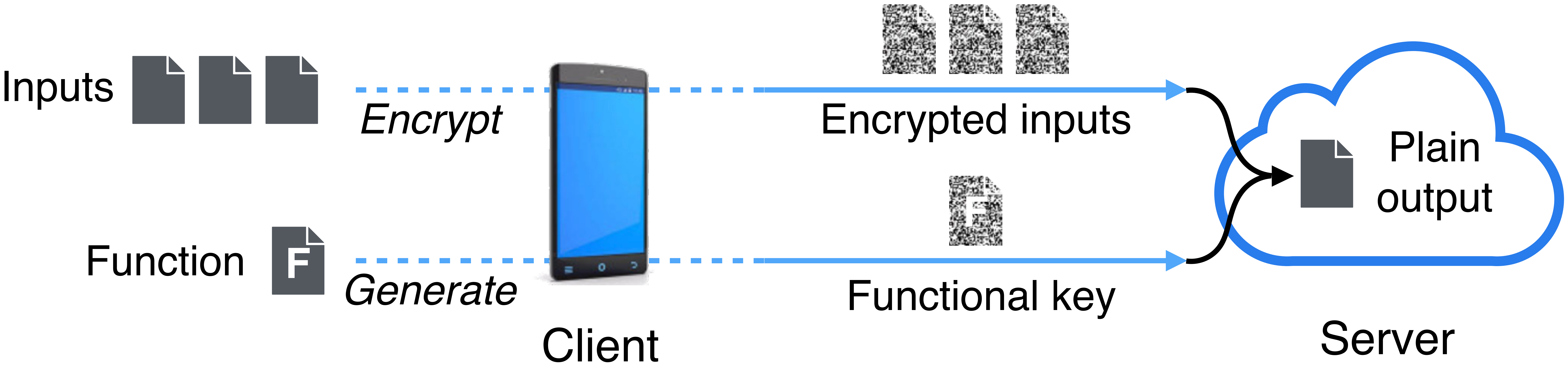
Security goal: **privacy** of data and queries.

Some perspective: computing on encrypted data

<u>SNARKs</u>: prove arbitrary statements on encrypted data.



<u>Fully Homomorphic Encryption</u>

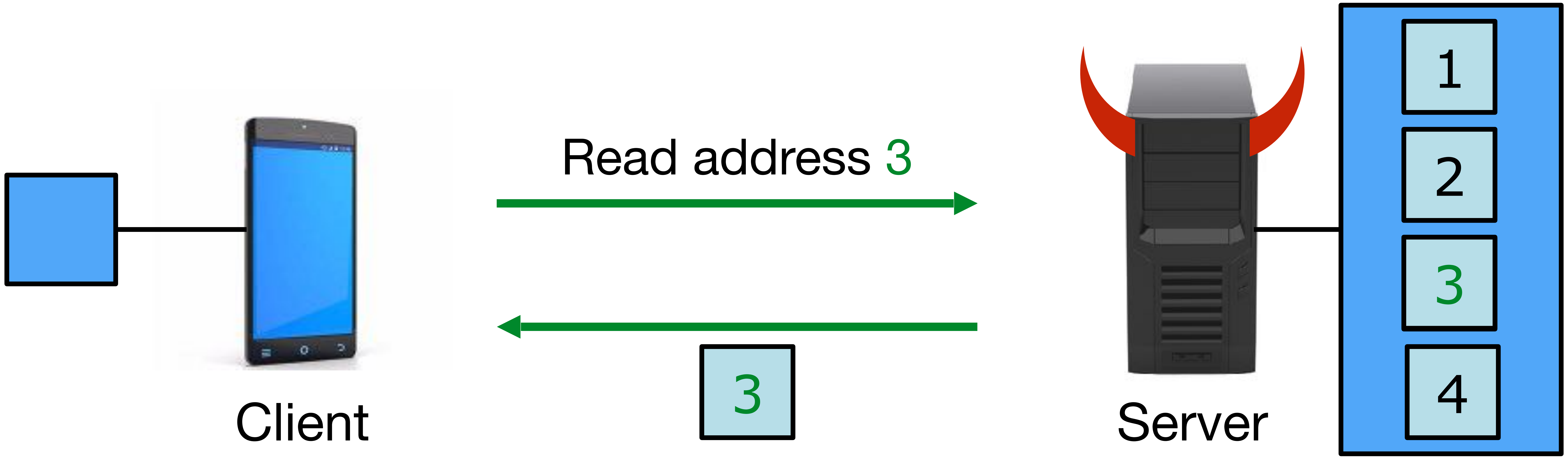


<u>Functional Encryption</u>

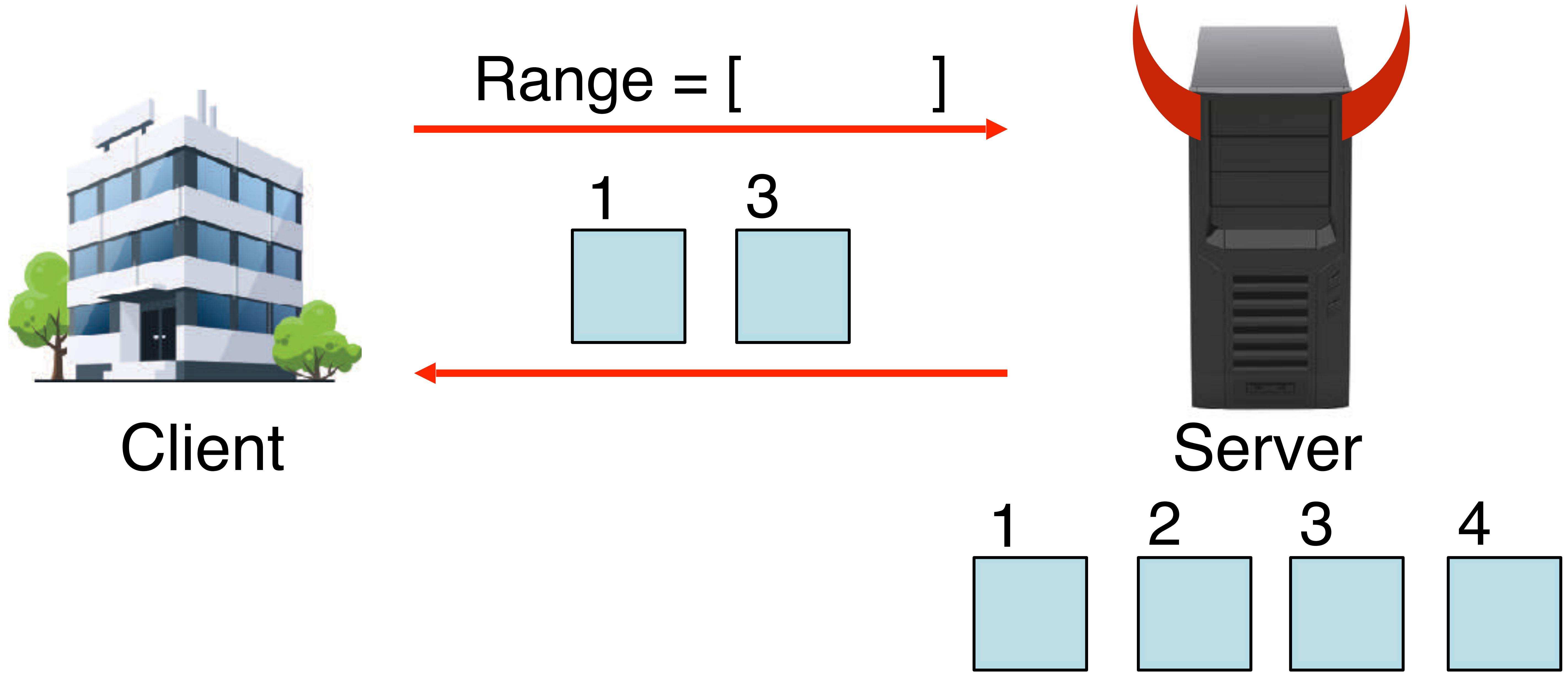
Is it okay to leak access pattern?

(No.)

Does leaking access pattern matter?



Example: range queries



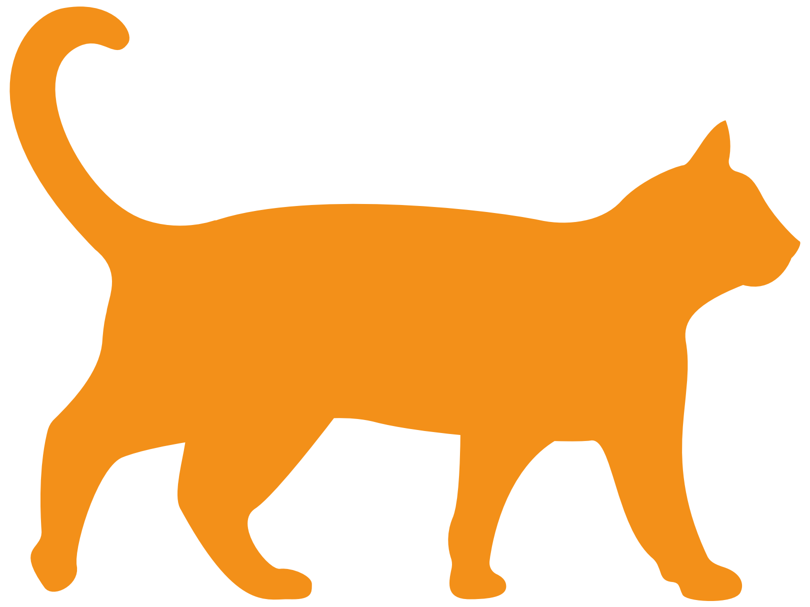
Imagine hospital storing patient information.

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.

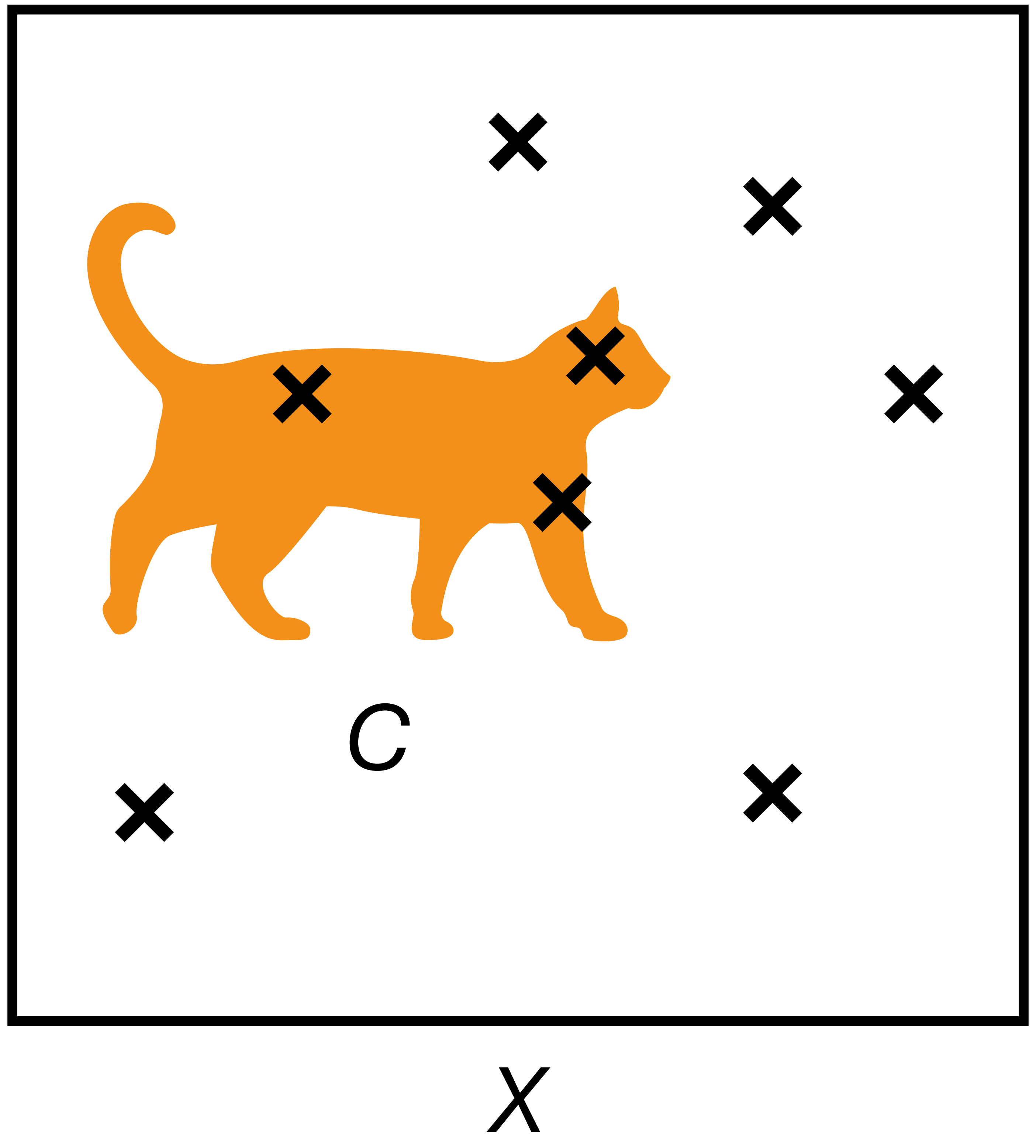
Fairly easy to state/use.

(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*.



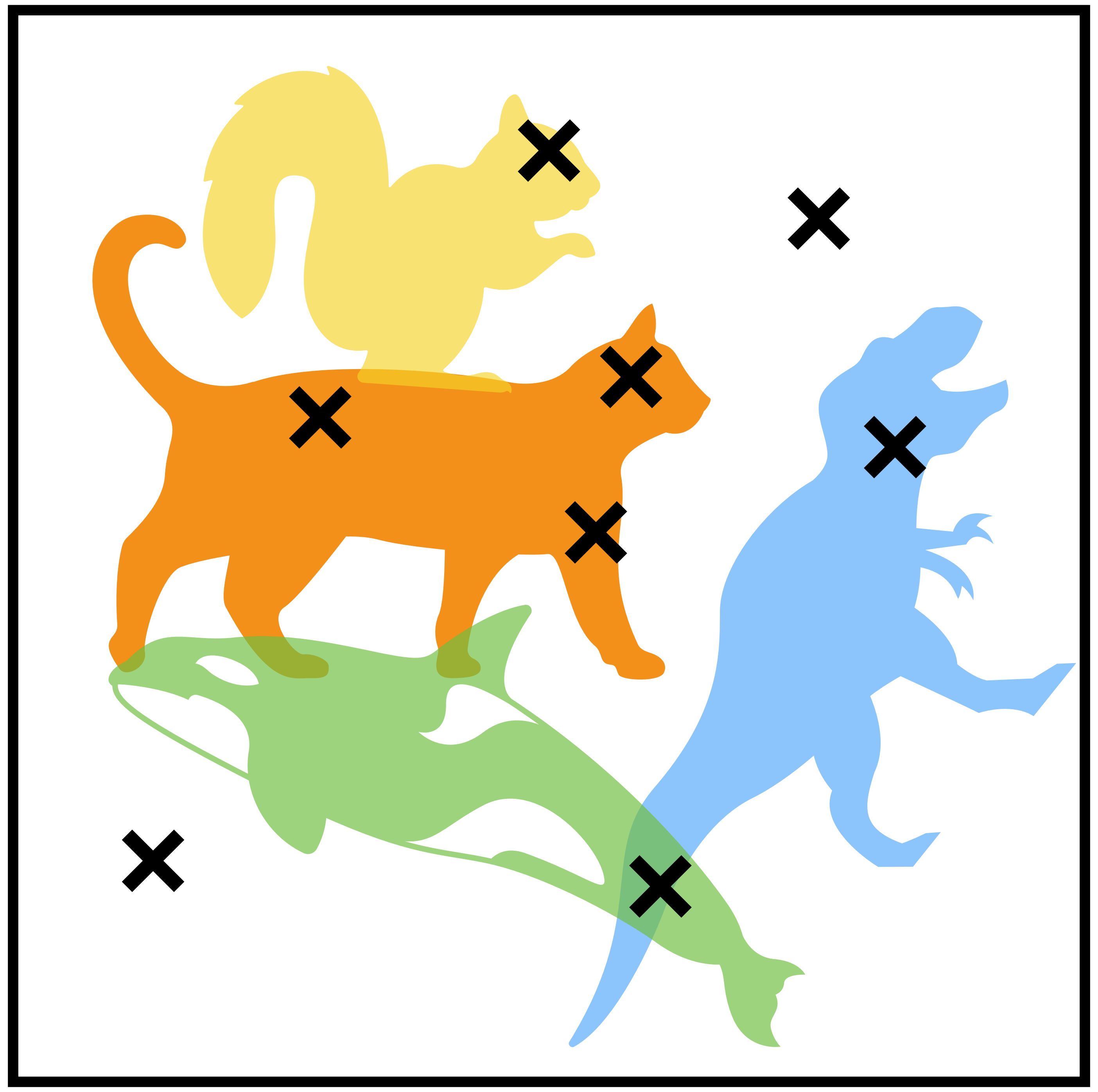
$$\Pr(C) \approx \frac{\#\text{points in } C}{\#\text{points total}}$$

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

Approximating a Concept Set

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

Goal: approximate their probabilities *simultaneously*.



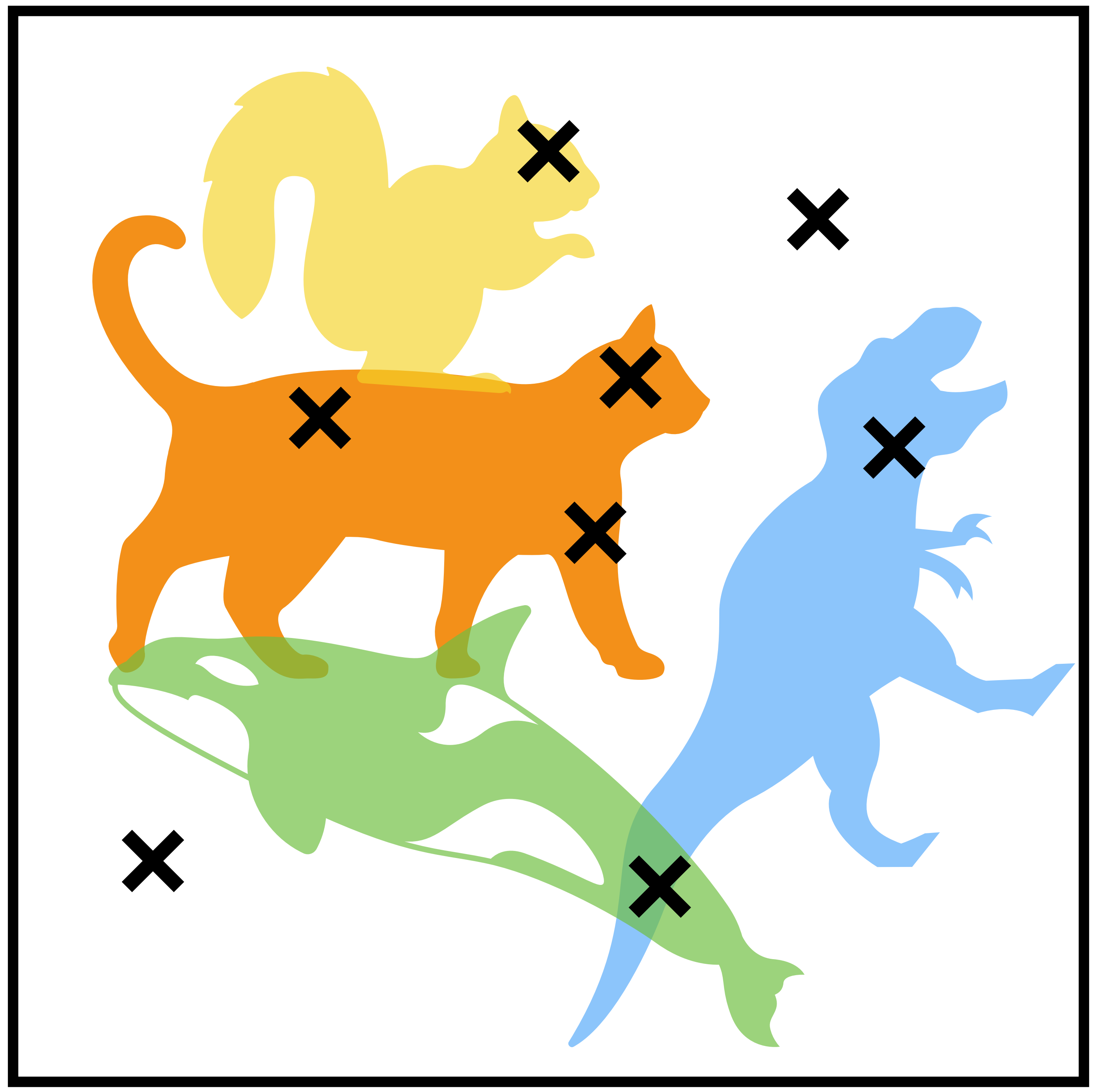
X

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

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

ε-sample Theorem

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



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

V & C 1971:

If $\mathfrak{C}$ 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 $|\mathfrak{C}|$*!*

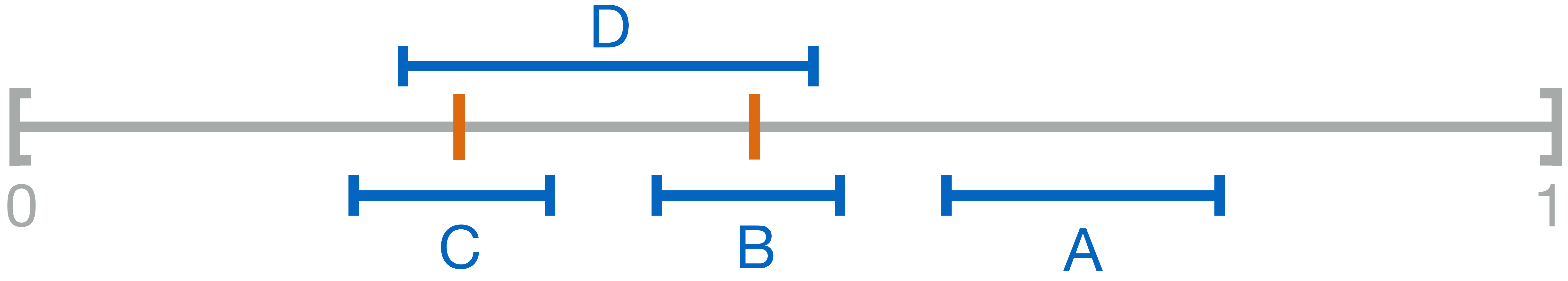
VC Dimension

Remaining Q: *what is the VC dimension?*

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

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

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



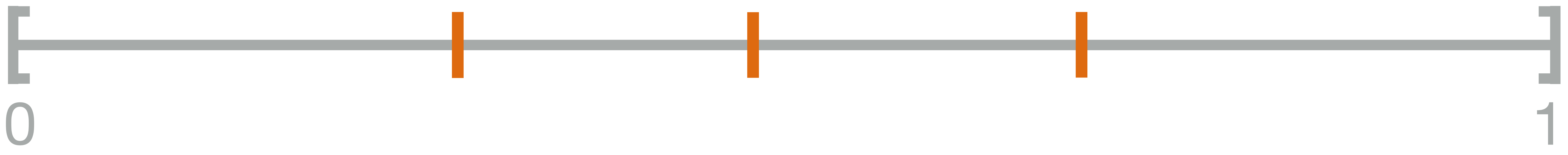
Subsets:

✕	✕	**OK**. Range A.
✕	O	**OK**. Range B.
O	✕	**OK**. Range C.
O	O	**OK**. Range D.

2 points = SHATTERED

VC Dimension

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



Subset: ○ × ○ Problem.

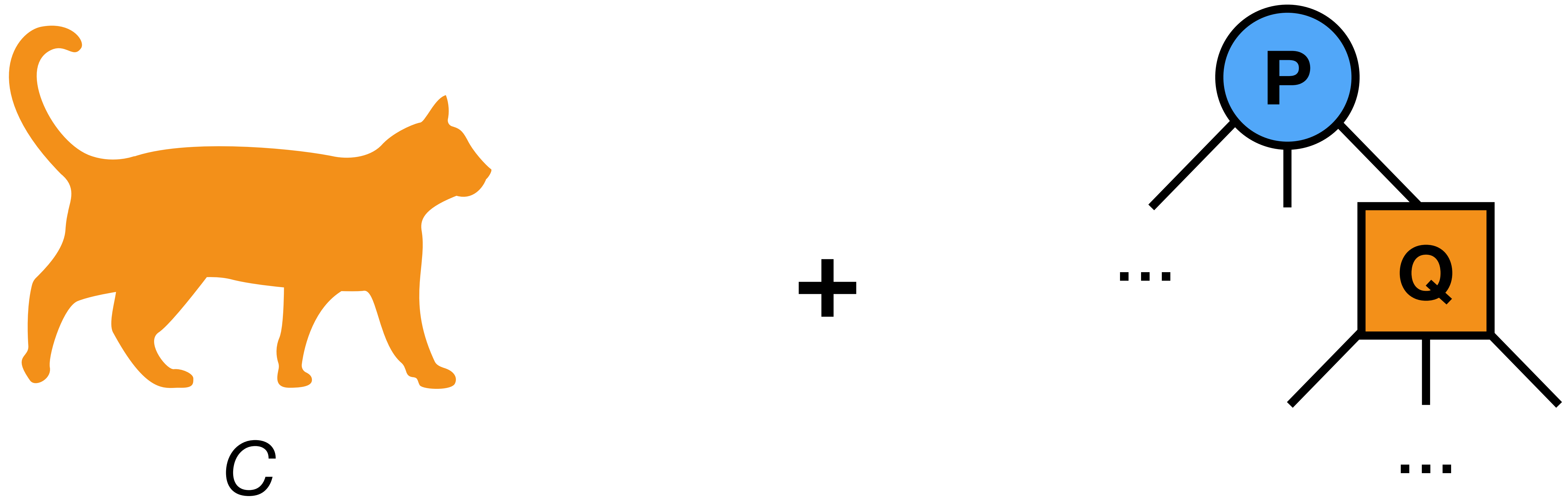
3 points = NOT SHATTERED

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

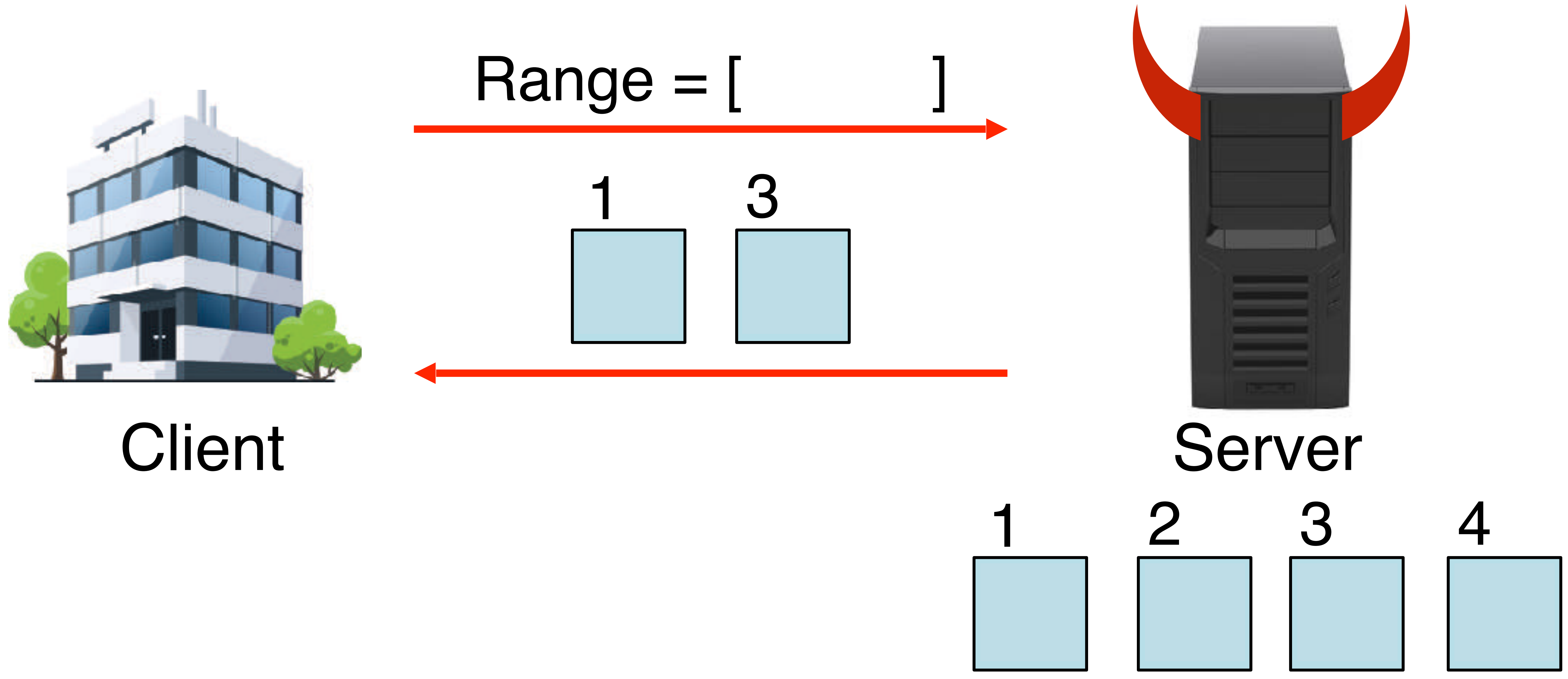
E.g. VC dimension of ranges is 2.

What typically matters is just that VC dim is finite.

Order Reconstruction



Problem Statement



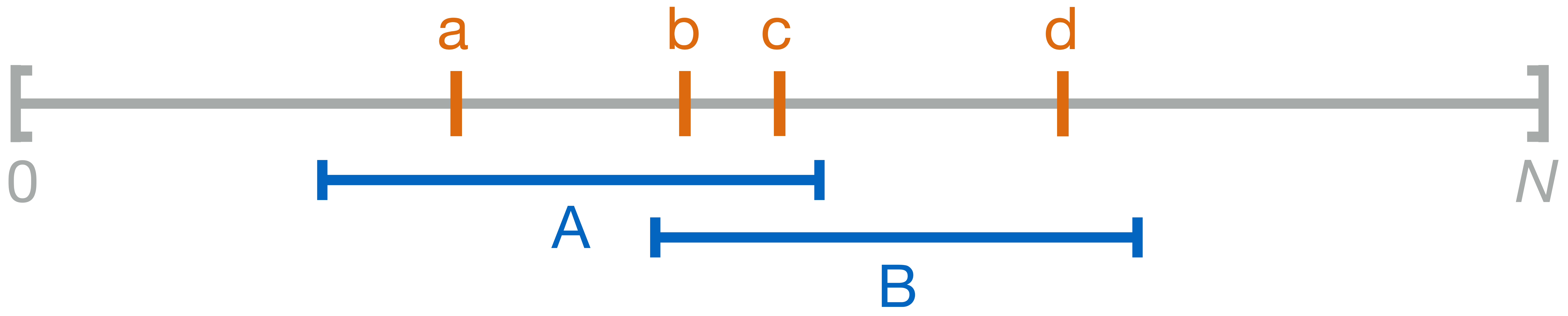
What can the server learn from the above leakage?

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.

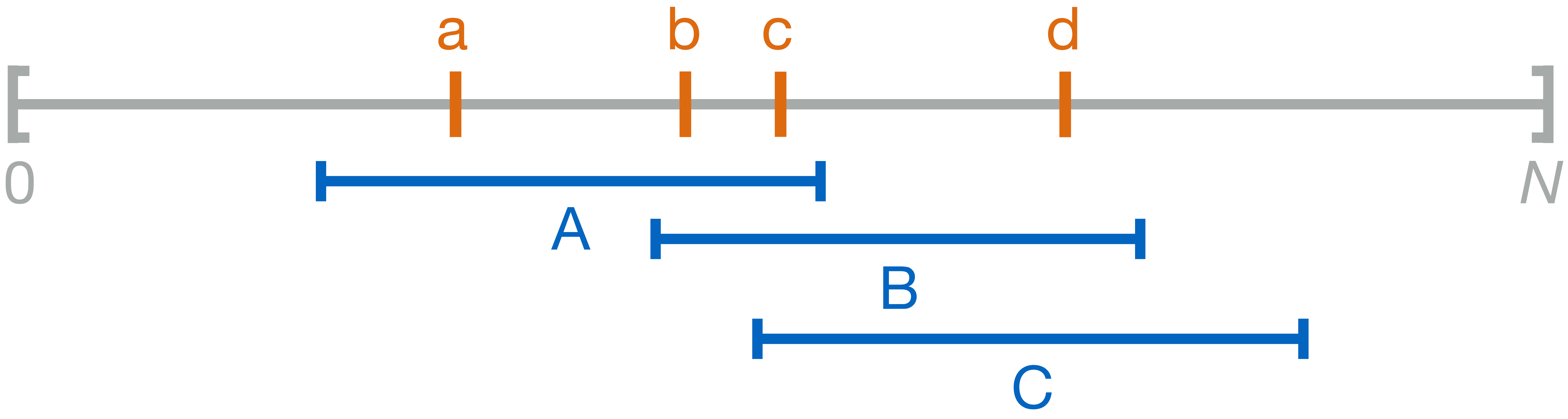
We learn something about the **order** of records.

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

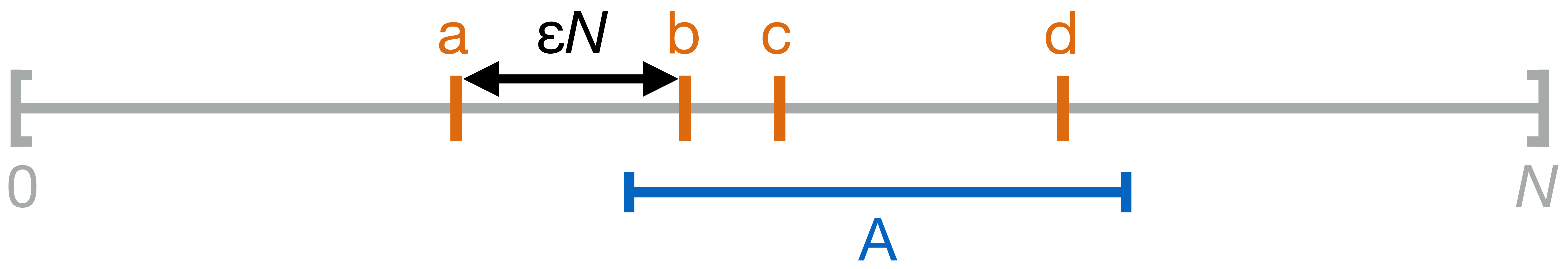
Strongest goal: **full** database reconstruction = recovering the exact value of every record.

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

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



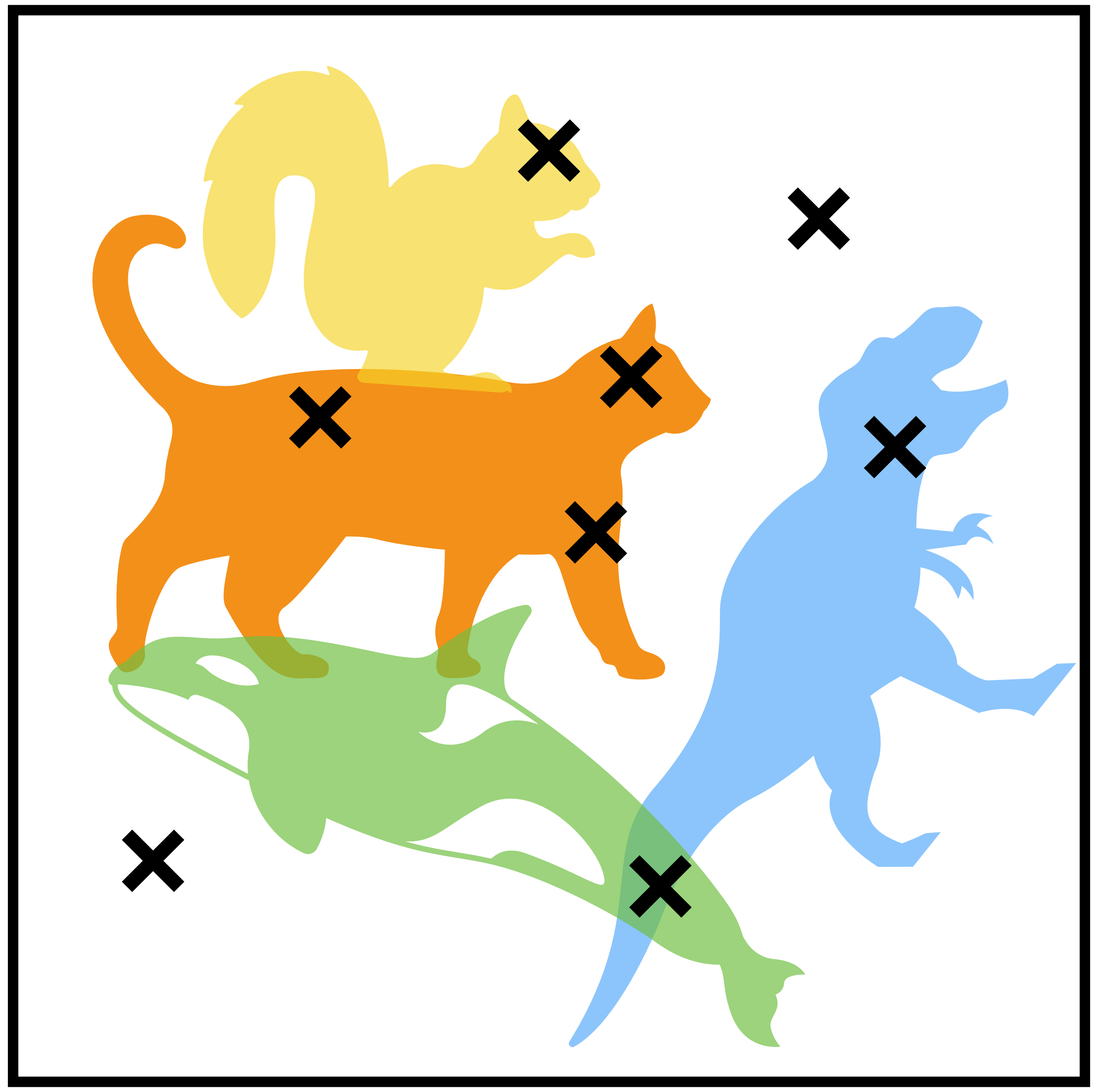
Intuition: if no query has an endpoint between a and b, then a and b can't be separated.

→ ε-approximate reconstruction is impossible.

You want a query endpoint to hit every interval ≥ ε*N*.
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.

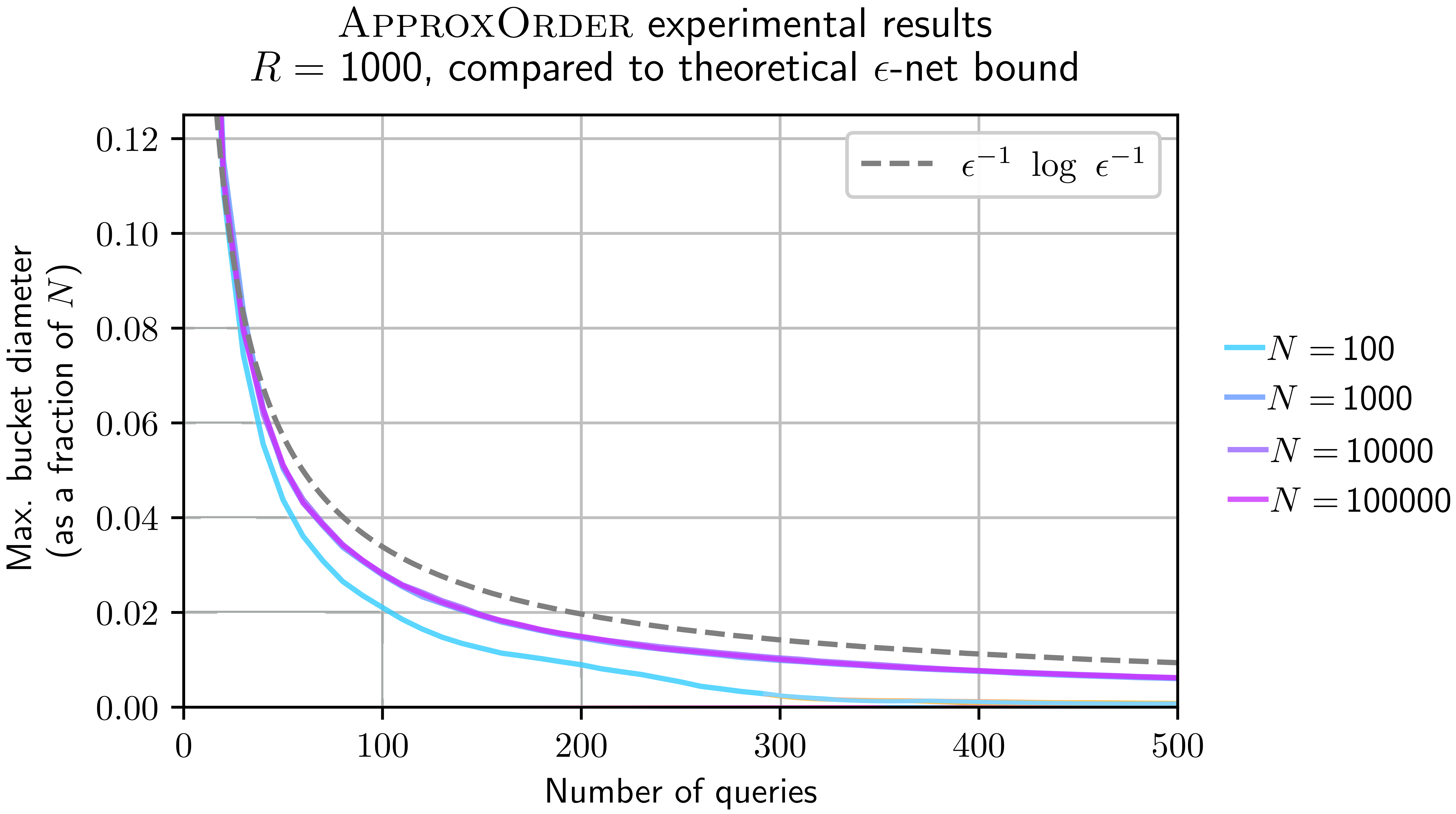
Very rough summary :

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**.

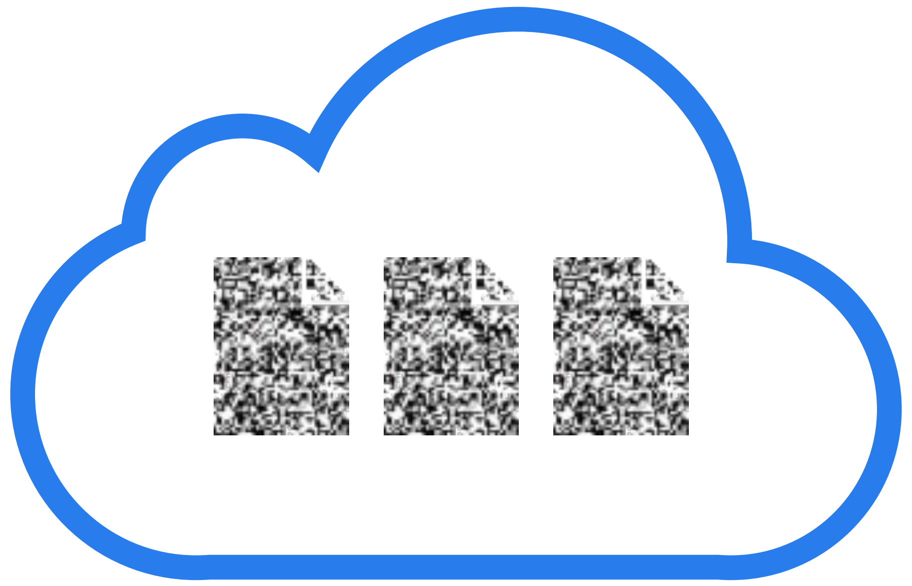
If adversary can observe *cache misses* from access to AES S-box tables, **also broken**.

Two issues:

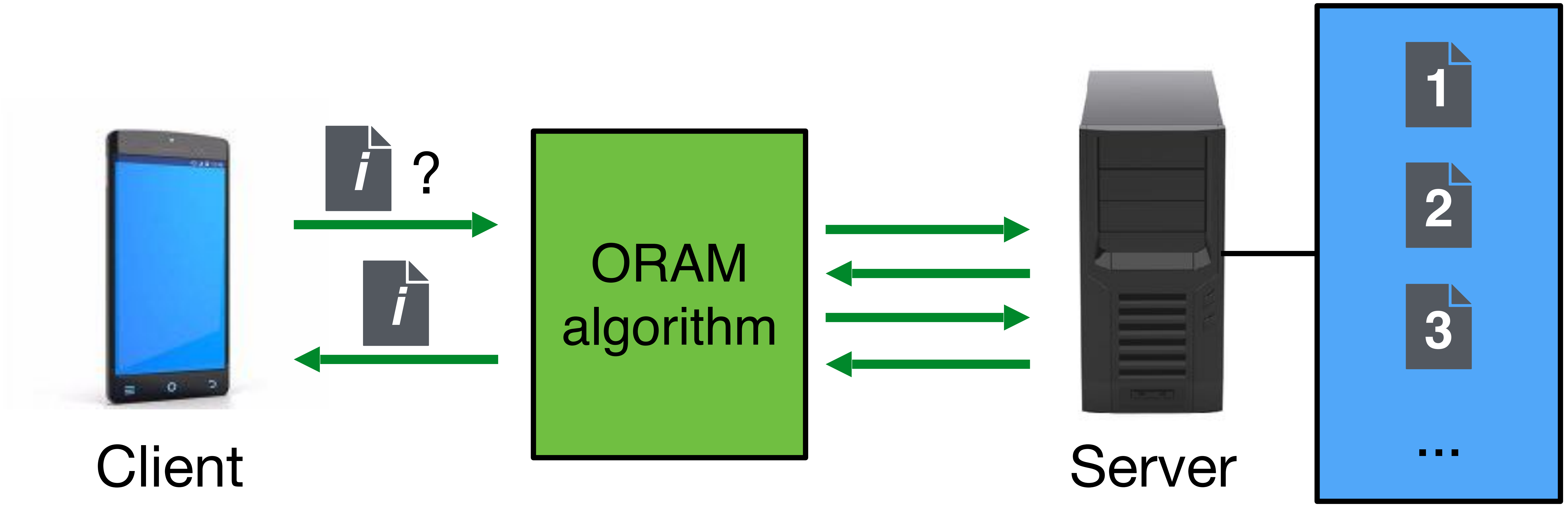
- Leaking access pattern can be (very) damaging.
- Many settings leak access pattern, completely or partially.

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

Oblivious algorithms



Magic Claim



Server stores *N* items.

Client fetches item *i*.

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

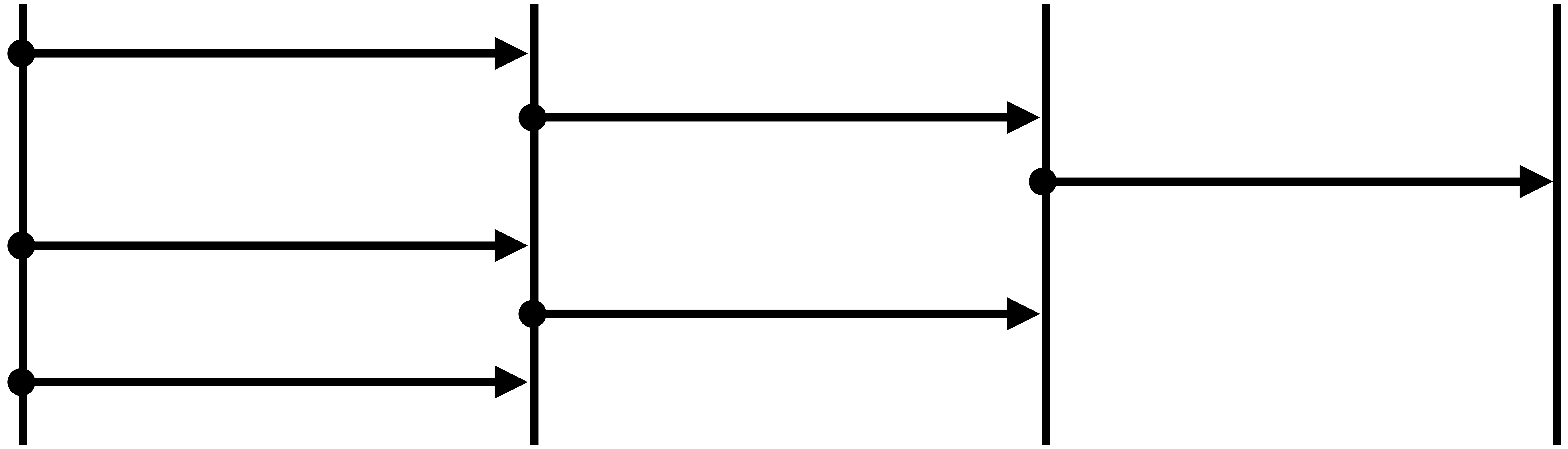
Efficiency: algorithm only queries O(log *N*) files.

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.

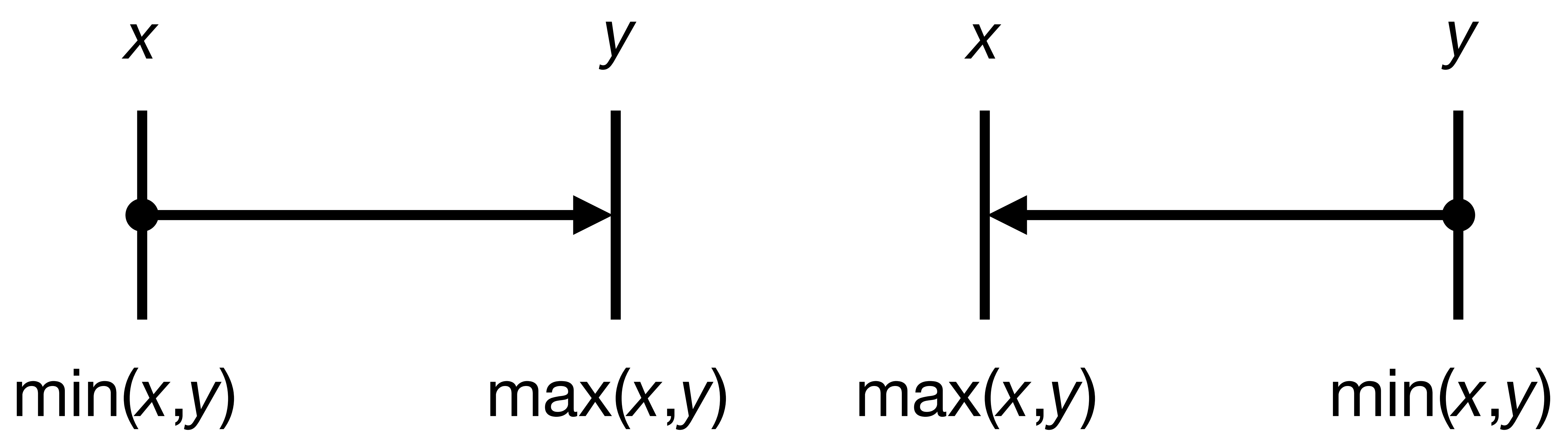
Which of the following algorithms are oblivious?
(assuming inputs are arrays of fixed size.)

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

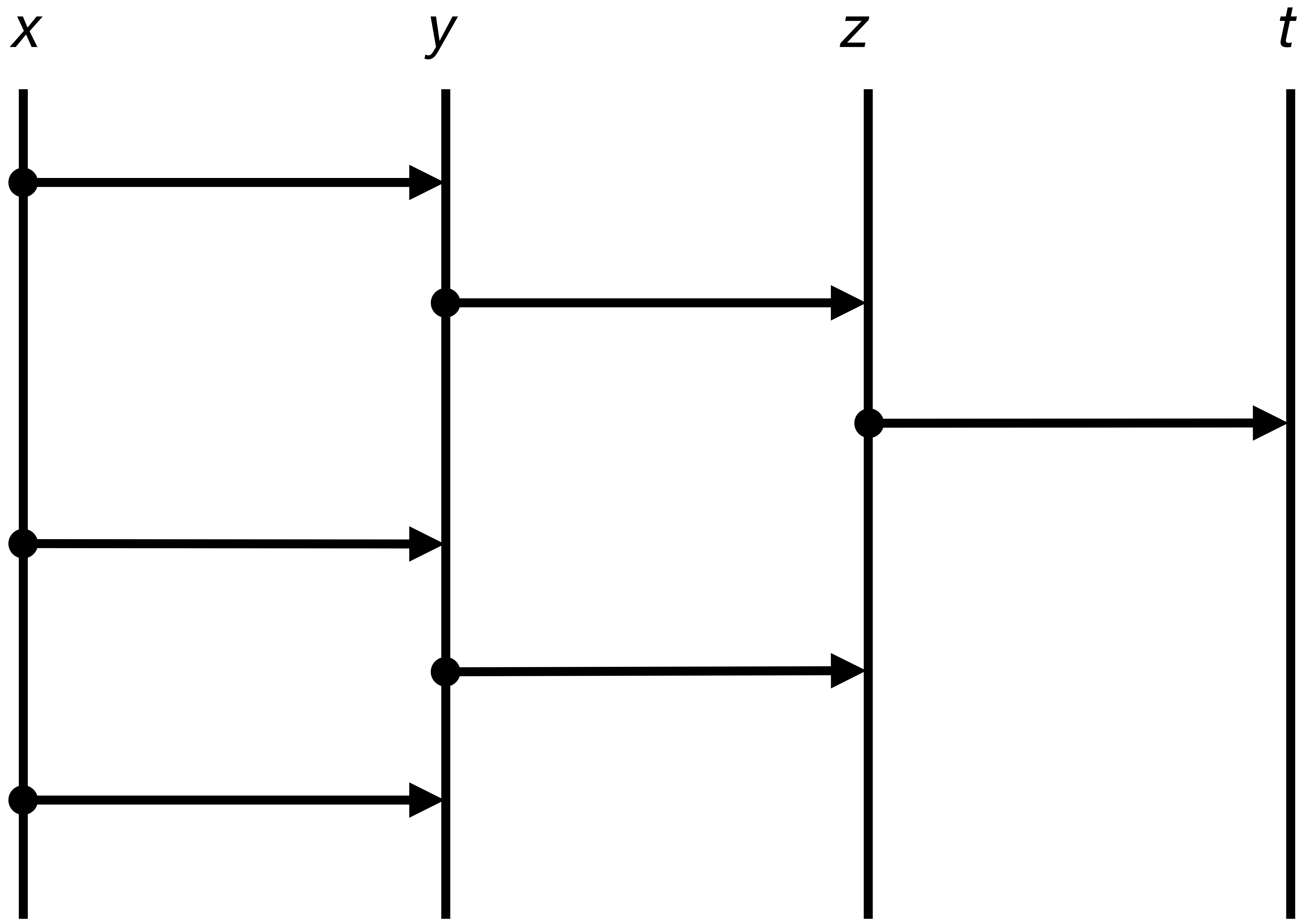
Sorting obliviously

Basic operation: sorting two elements.

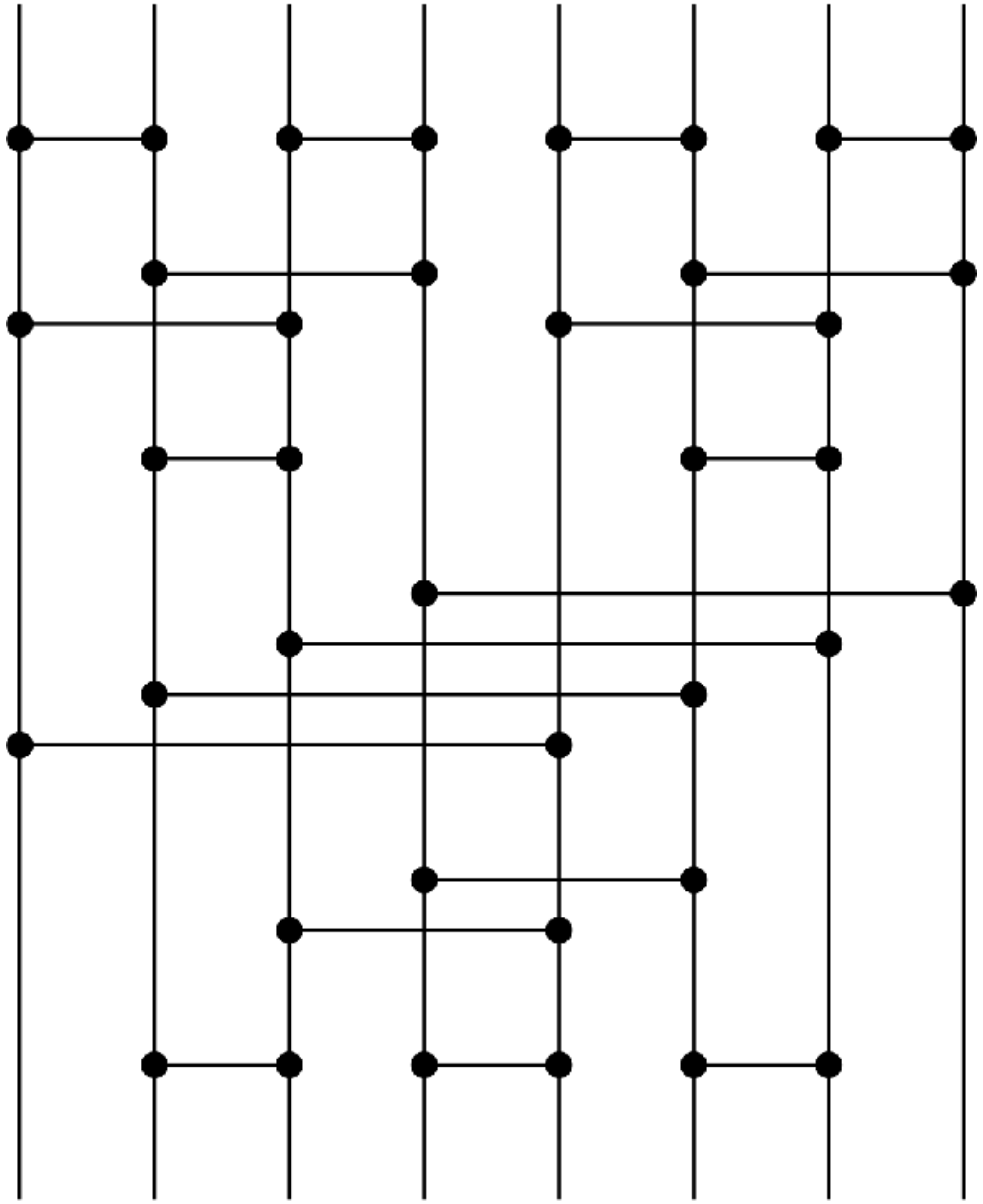
Compare and swap: on input (x,y), if x < y, output (x,y), else output (y,x).



Bubble Sort



Batcher's sort



Sorting network of size $O(n \log^2 n)$ that correctly sorts all inputs.

Oblivious Sorting: conclusion

Batcher's sort: practical sorting network of size $O(n \log^2 n)$.

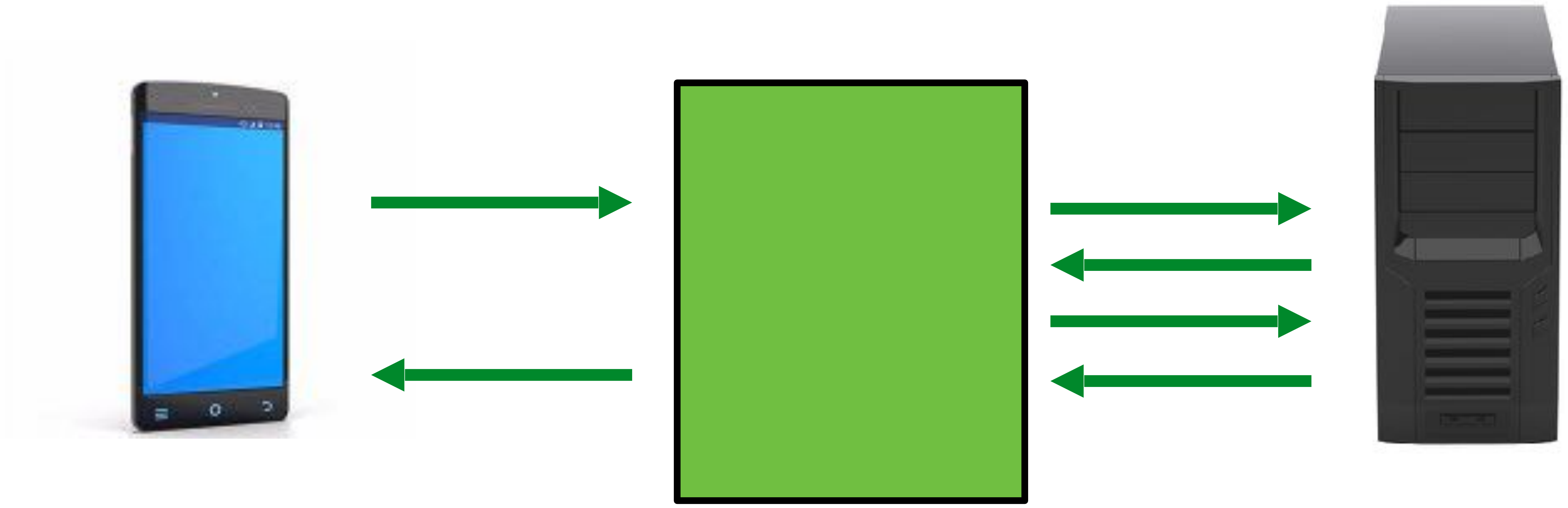
Bonus: in a parallel computation model, only need $O(\log^2 n)$ steps.

→ 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.

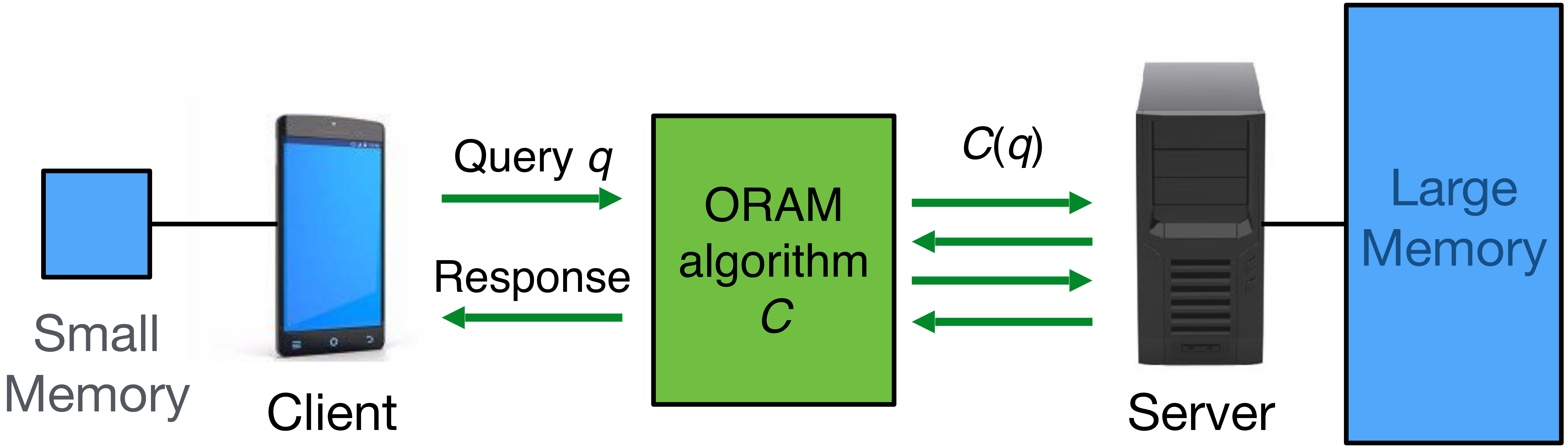
Can we do this generically?

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

This is what Oblivious RAM (ORAM) does.

Disclaimer: does not hide number of accesses.

Oblivious RAM

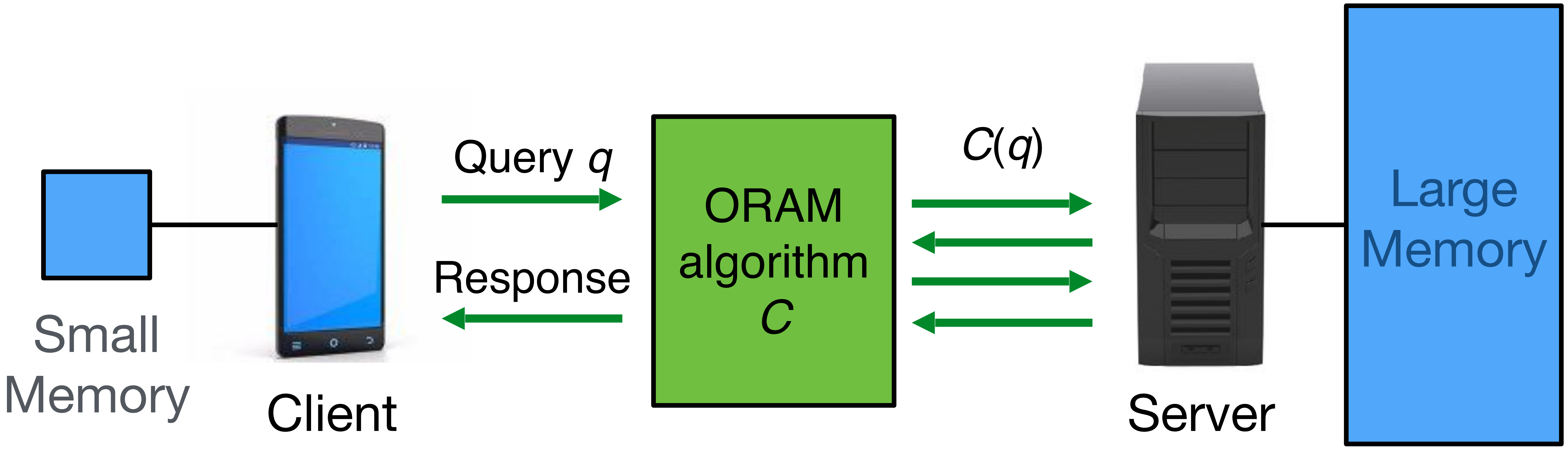


Client wants to do queries q_1, q_2, …, q_n.

Each q_i is either:

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

Oblivious RAM, cont'd

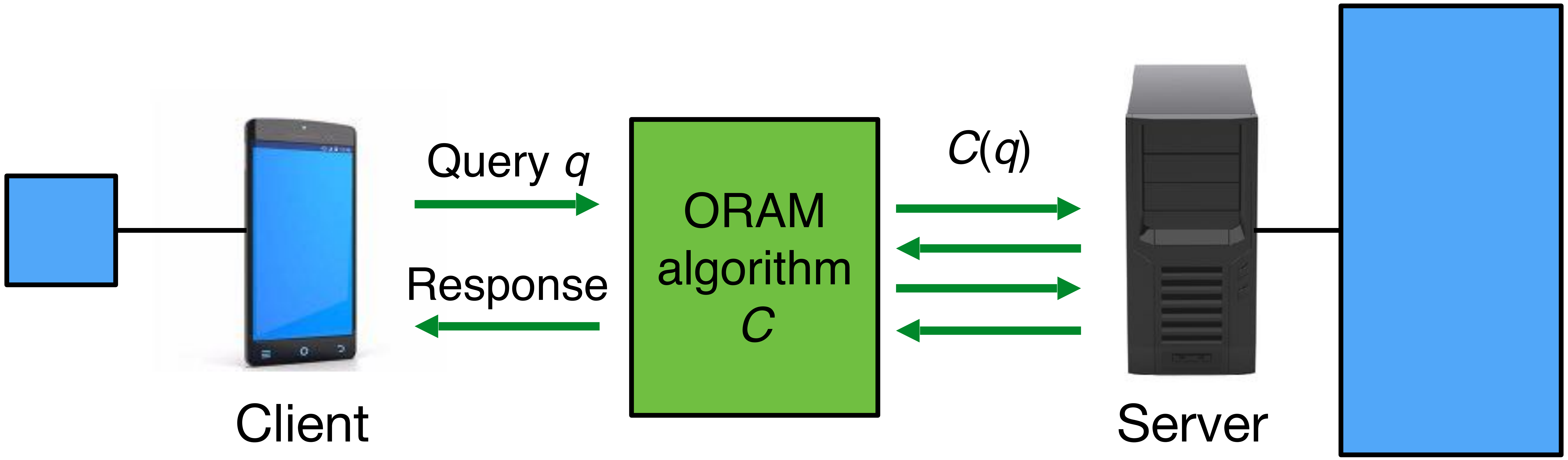


ORAM algorithm C (or ORAM “compiler”): transforms each query q by the client into one or several read/write queries $C(q)$ to server.

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.

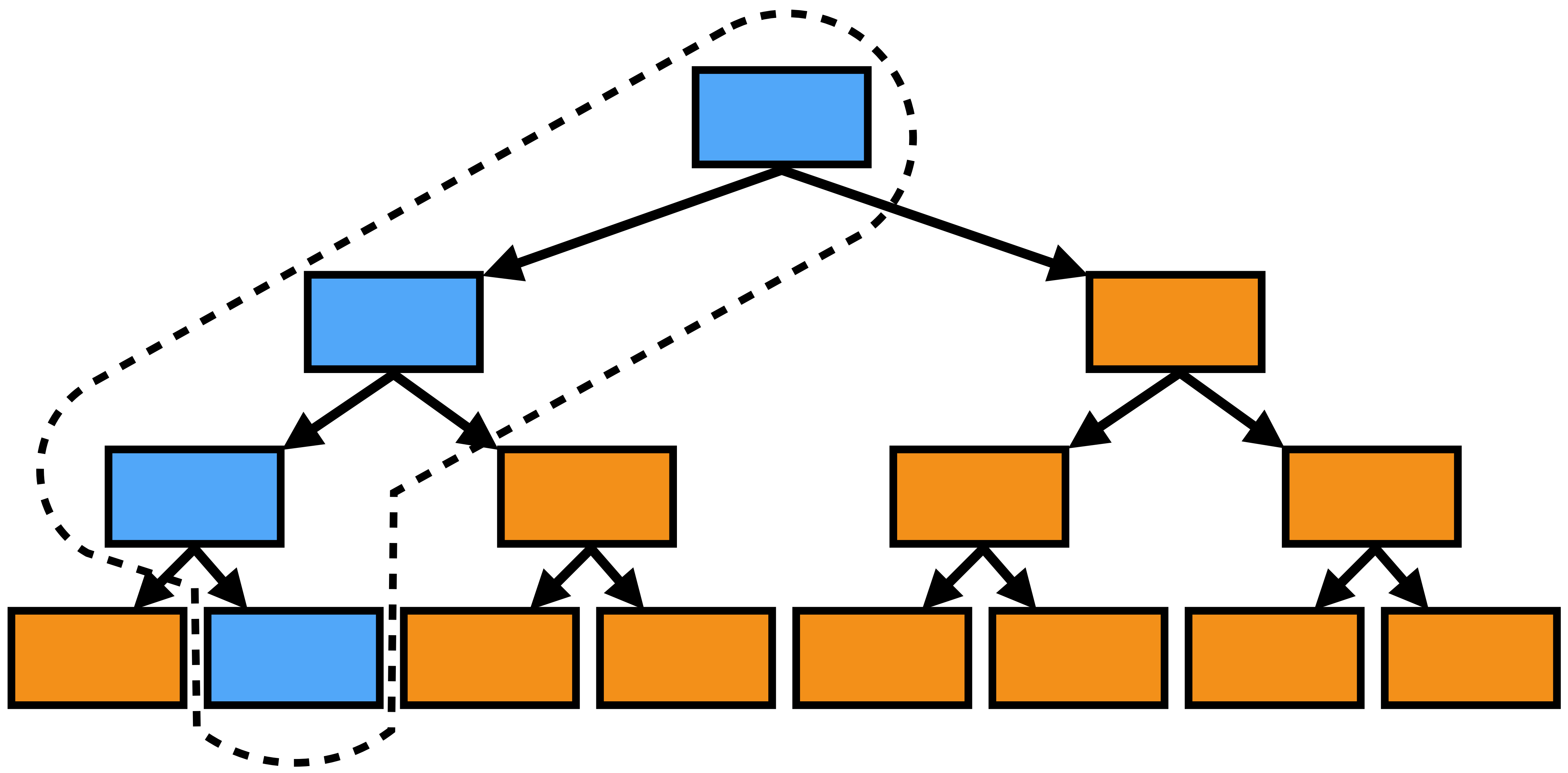
Security: trivial.

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).

Tree ORAM



Tree ORAM

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

What is actually used:

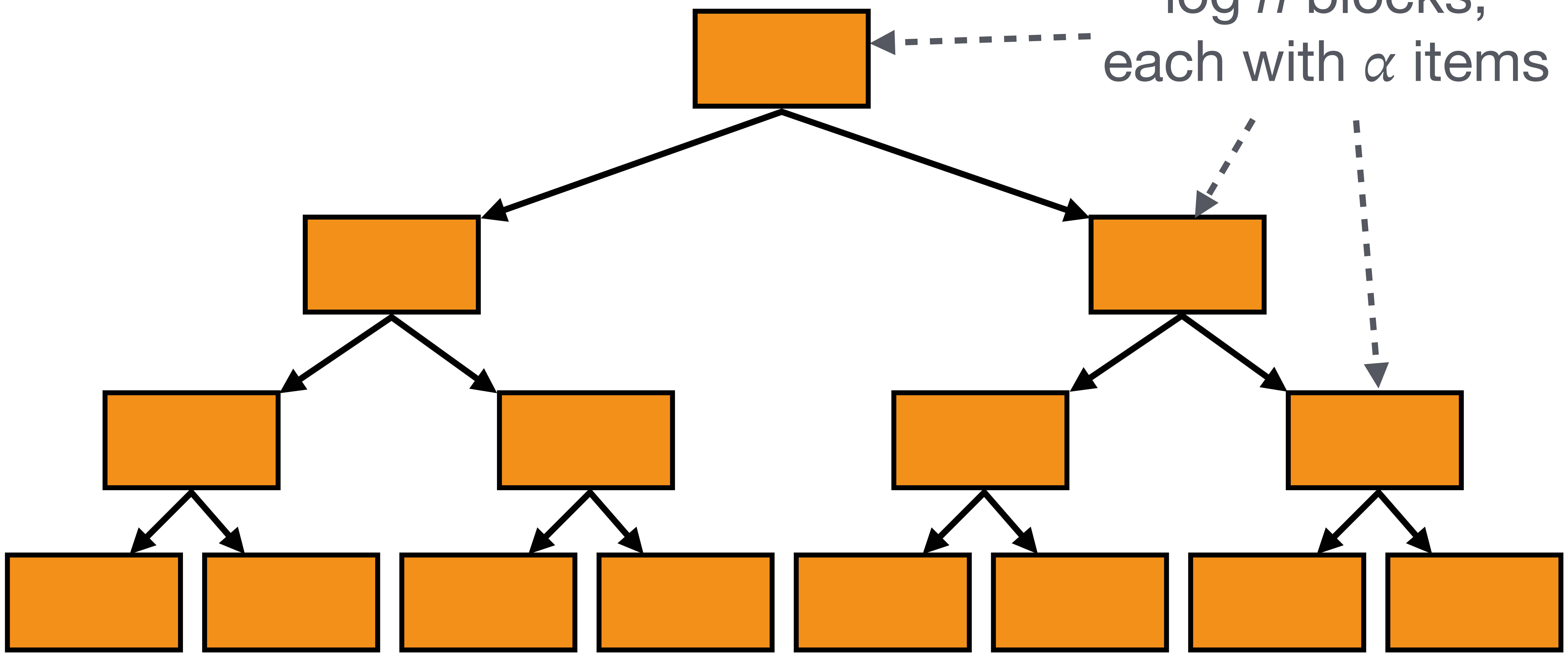
Tree ORAM *by Shi et al. '11*

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

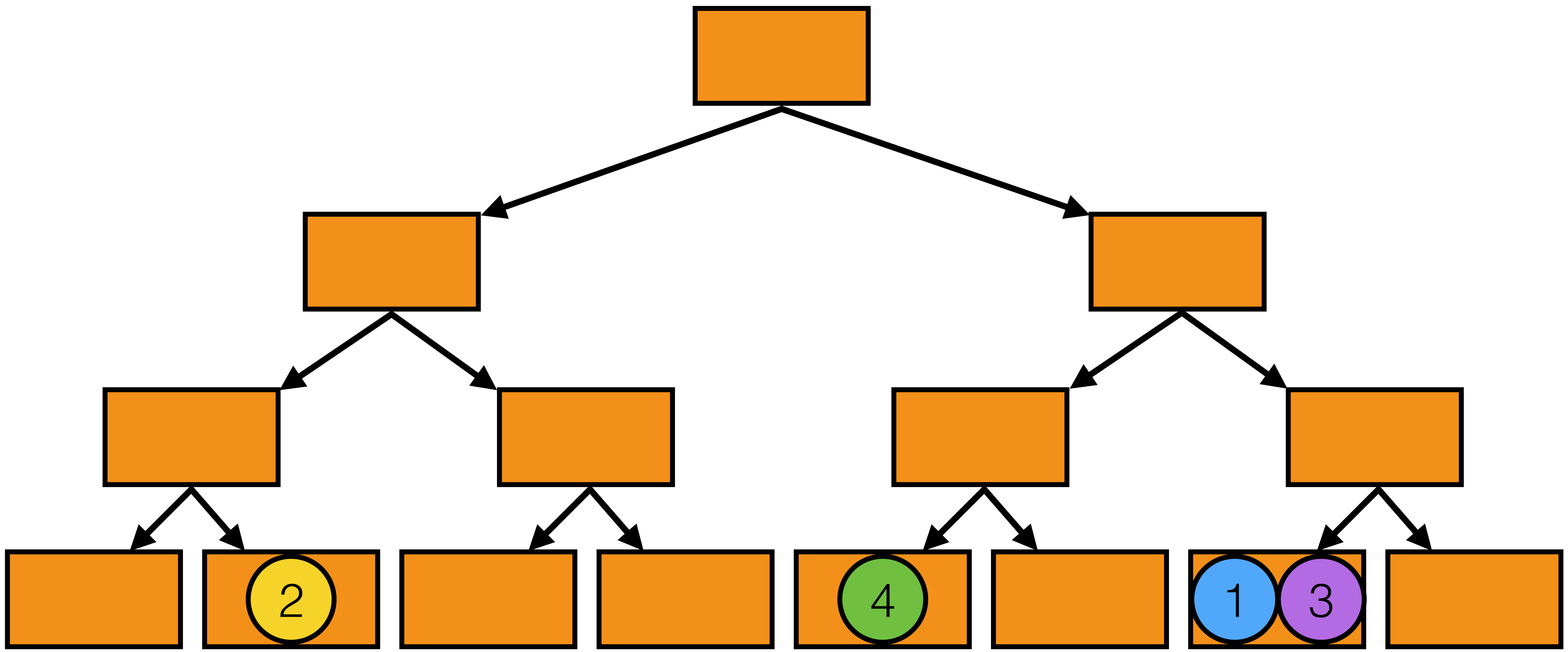


Server-side memory is a full binary tree with $\log(n/\alpha)$ levels.

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 $b = \lfloor i/\alpha \rfloor$.

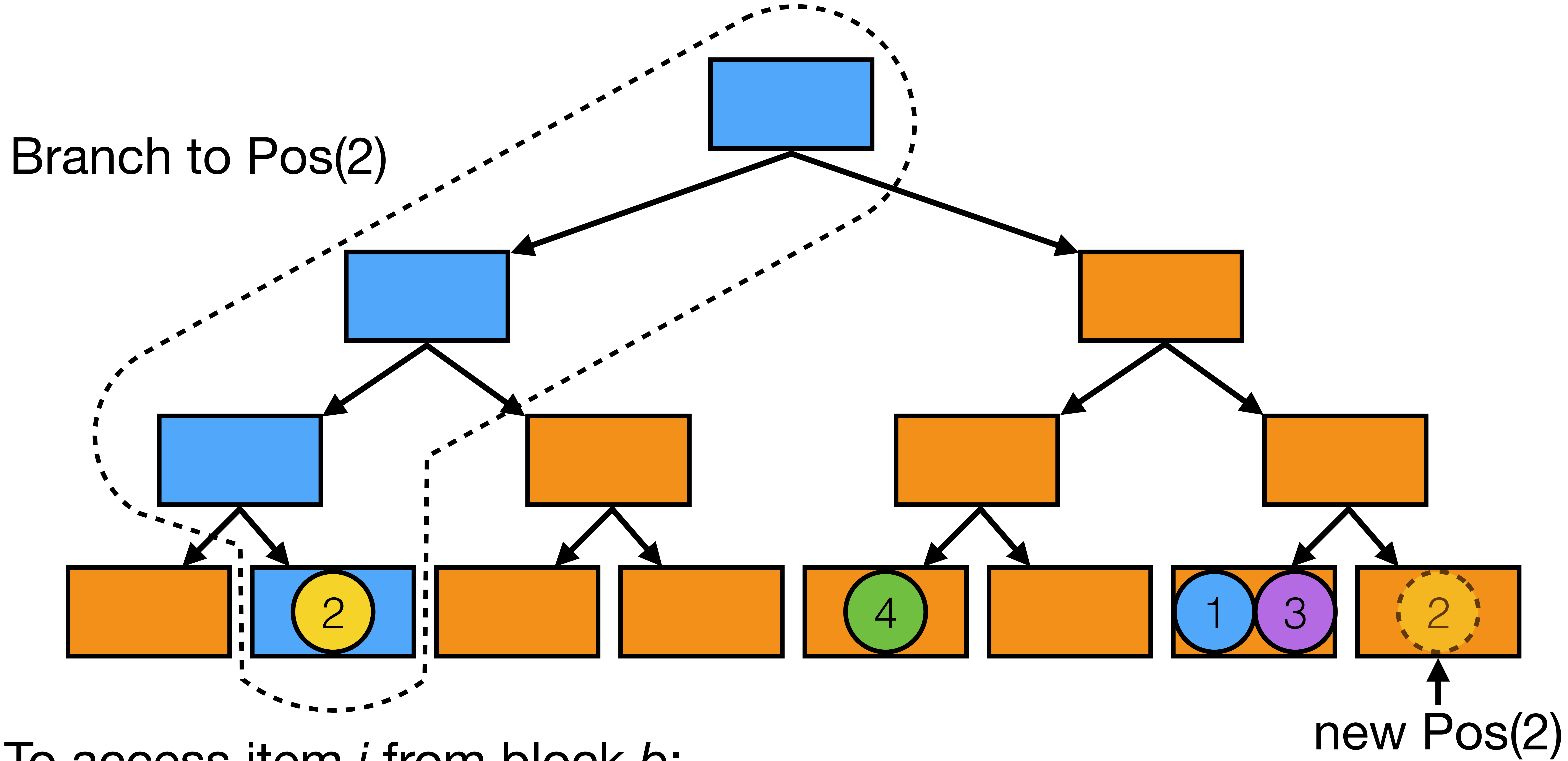
At start:

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*).

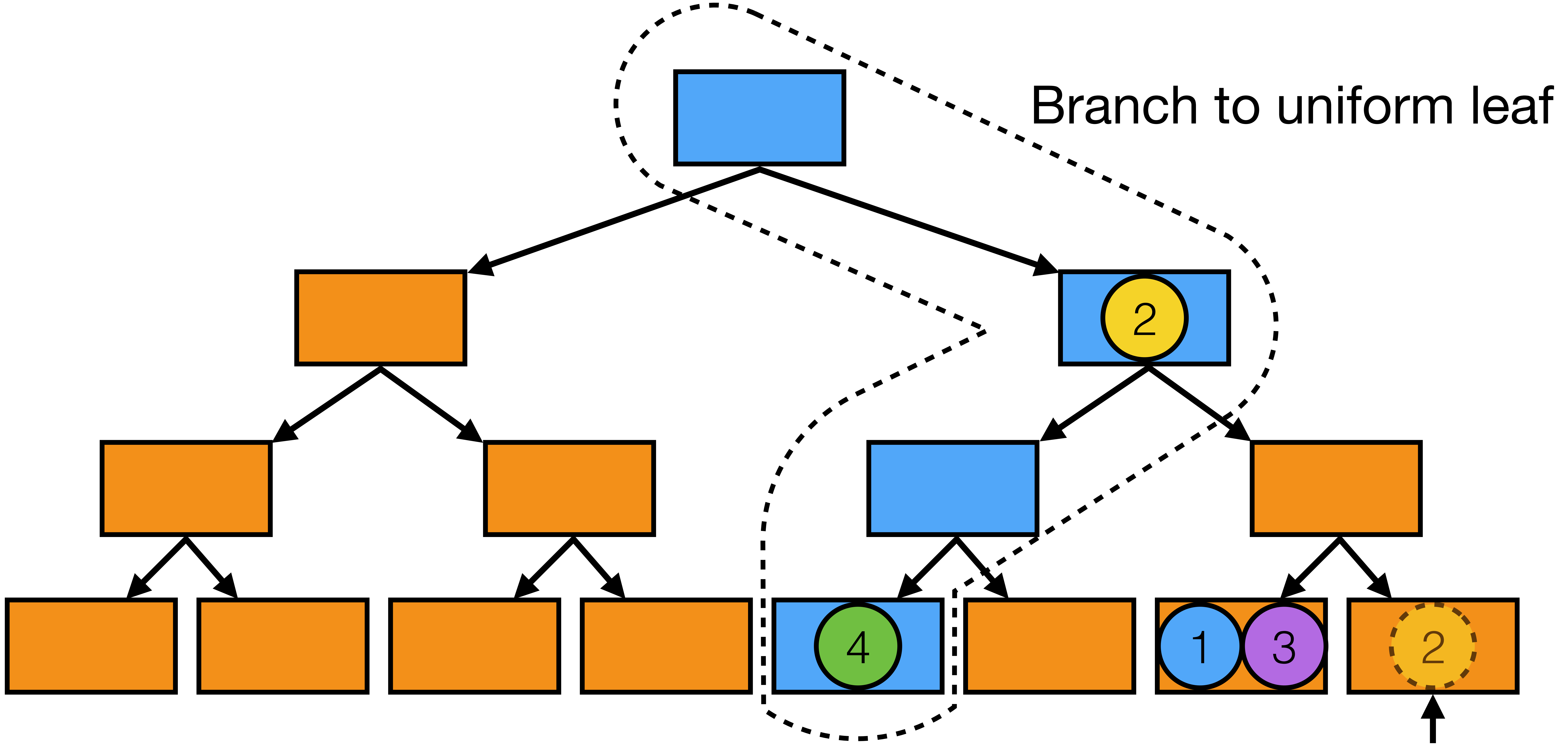
Lookup



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

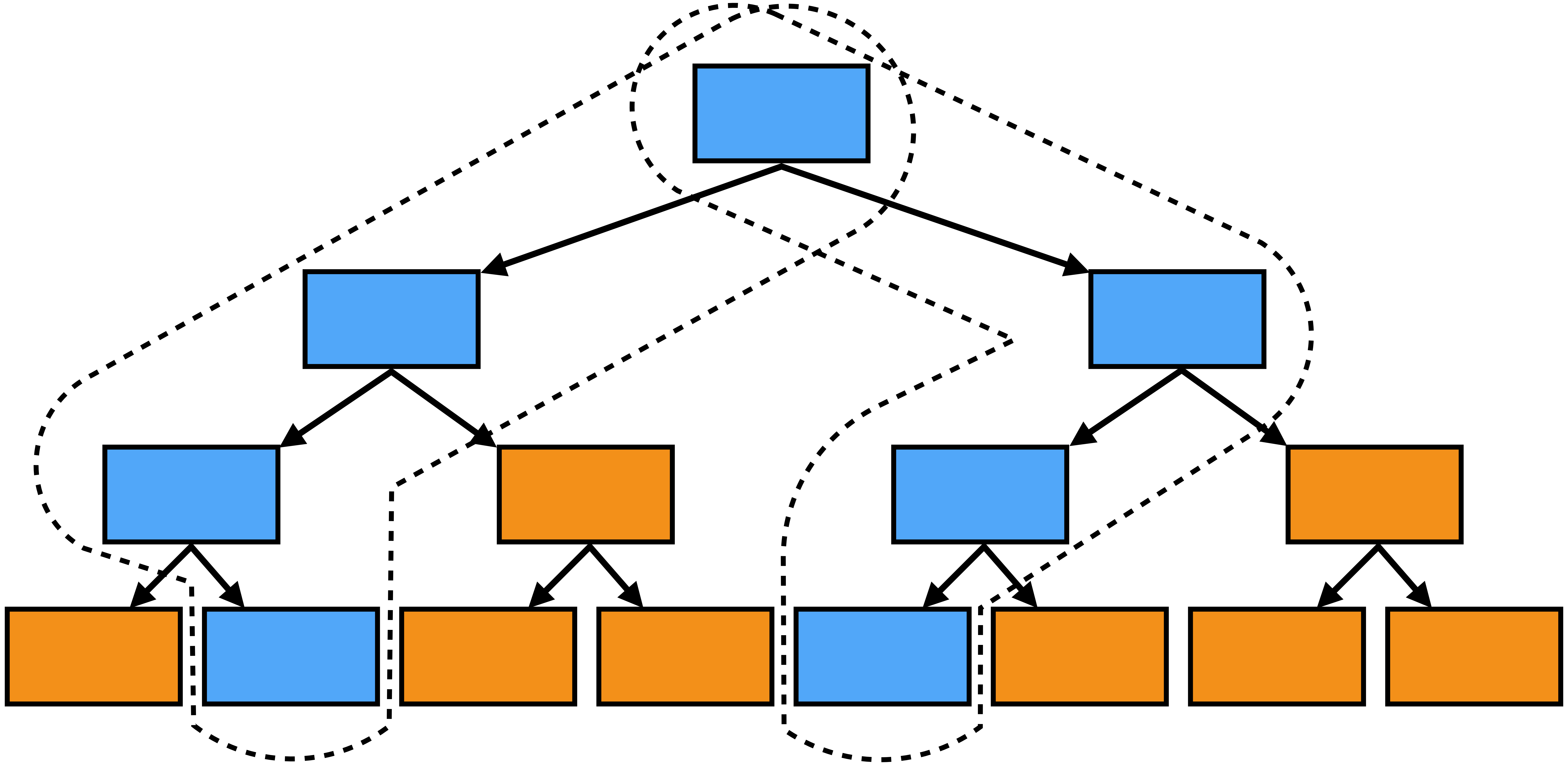
1. Pick branch to uniformly random leaf.
2. Push every block in the branch as far down as possible (preserving that block *b* must remain on branch to Pos(*b*)).

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.

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 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: $O(1)$.

Note: possible to combine ORAM with FHE and MPC.

In practice

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!

Fairly practical: used by Signal for contact discovery.

Signal