CS 350S: Privacy-Preserving Systems

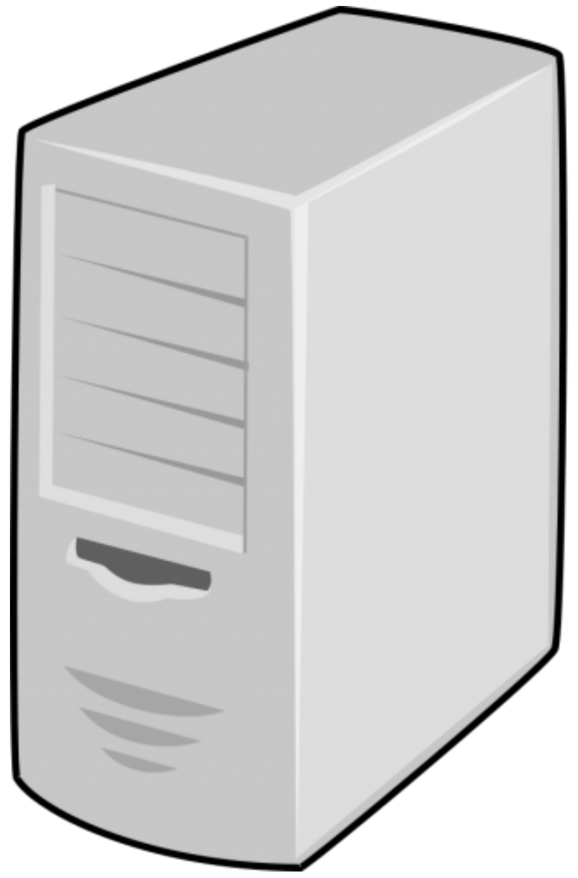
Oblivious RAM

Outline

1. Overview
2. Square-root construction
3. Hierarchical construction
4. Limitations
5. Student presentation: PathORAM

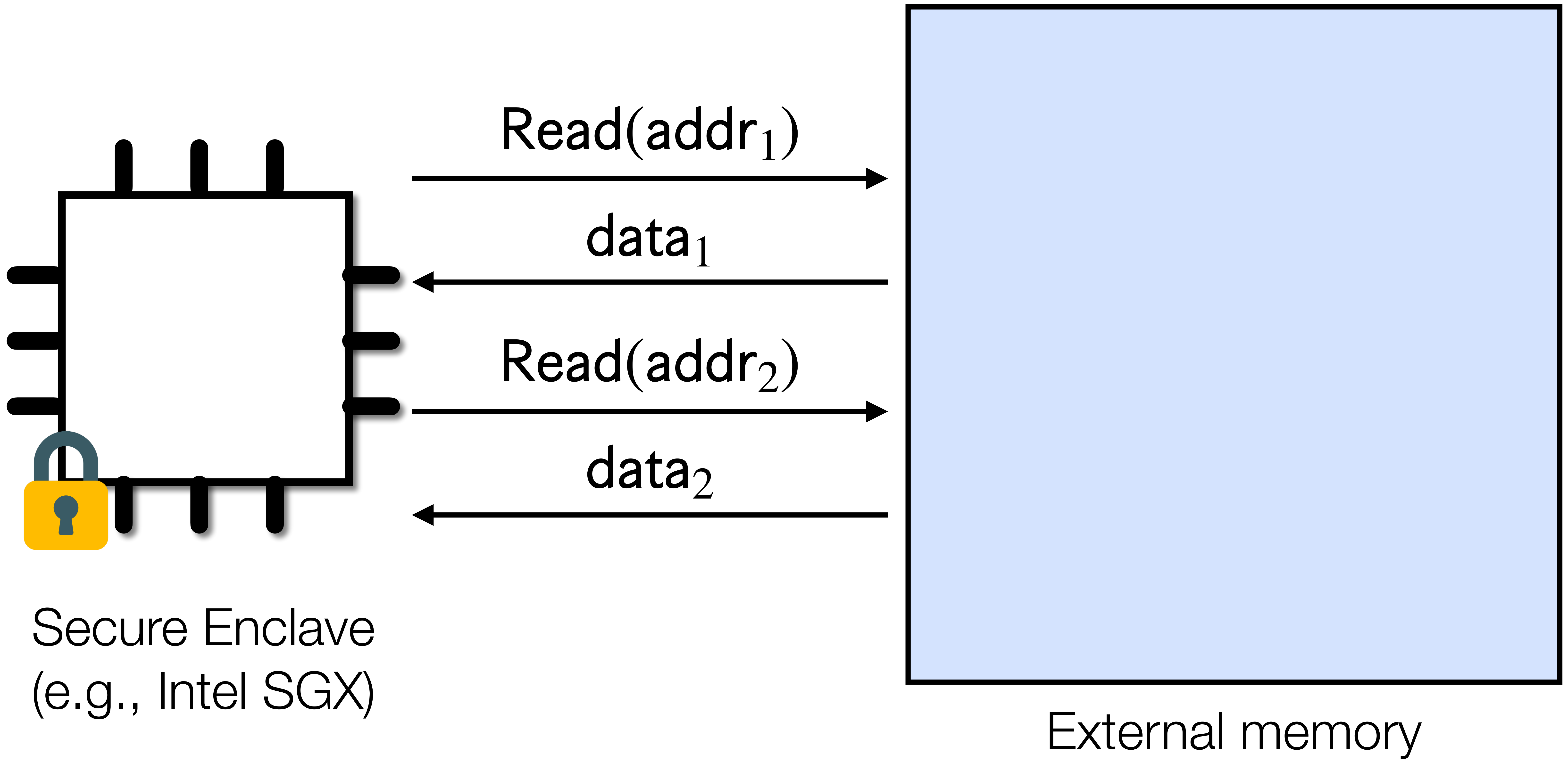
Access patterns can reveal sensitive information

$\mathrm{Enc}_k(\text{heart disease})$	$\mathrm{Enc}_k(\text{info}_1)$
$\mathrm{Enc}_k(\text{cancer})$	$\mathrm{Enc}_k(\text{info}_2)$
$\mathrm{Enc}_k(\text{diabetes})$	$\mathrm{Enc}_k(\text{info}_3)$
...	...

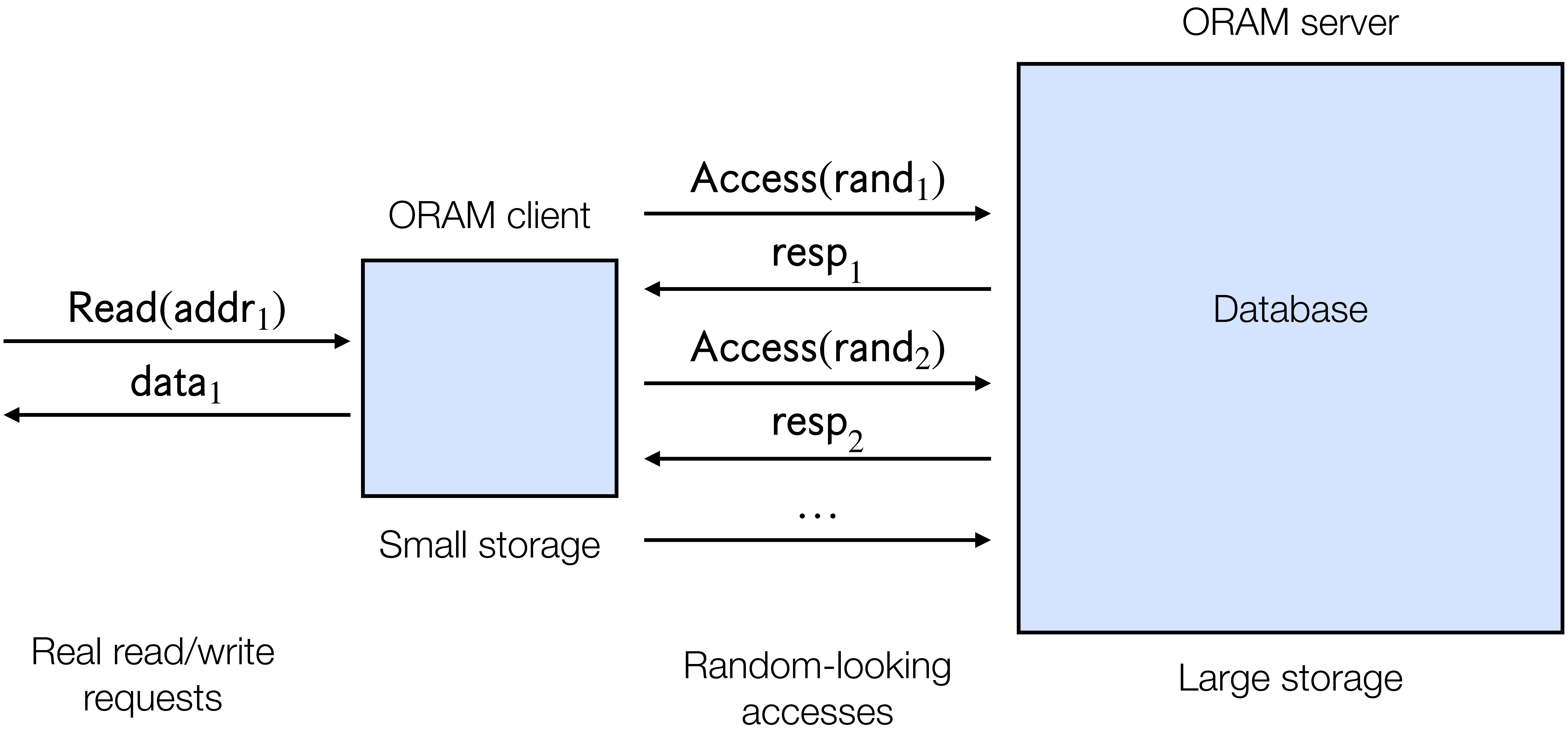


Attacker can infer that Alice and Bob have the same medical condition

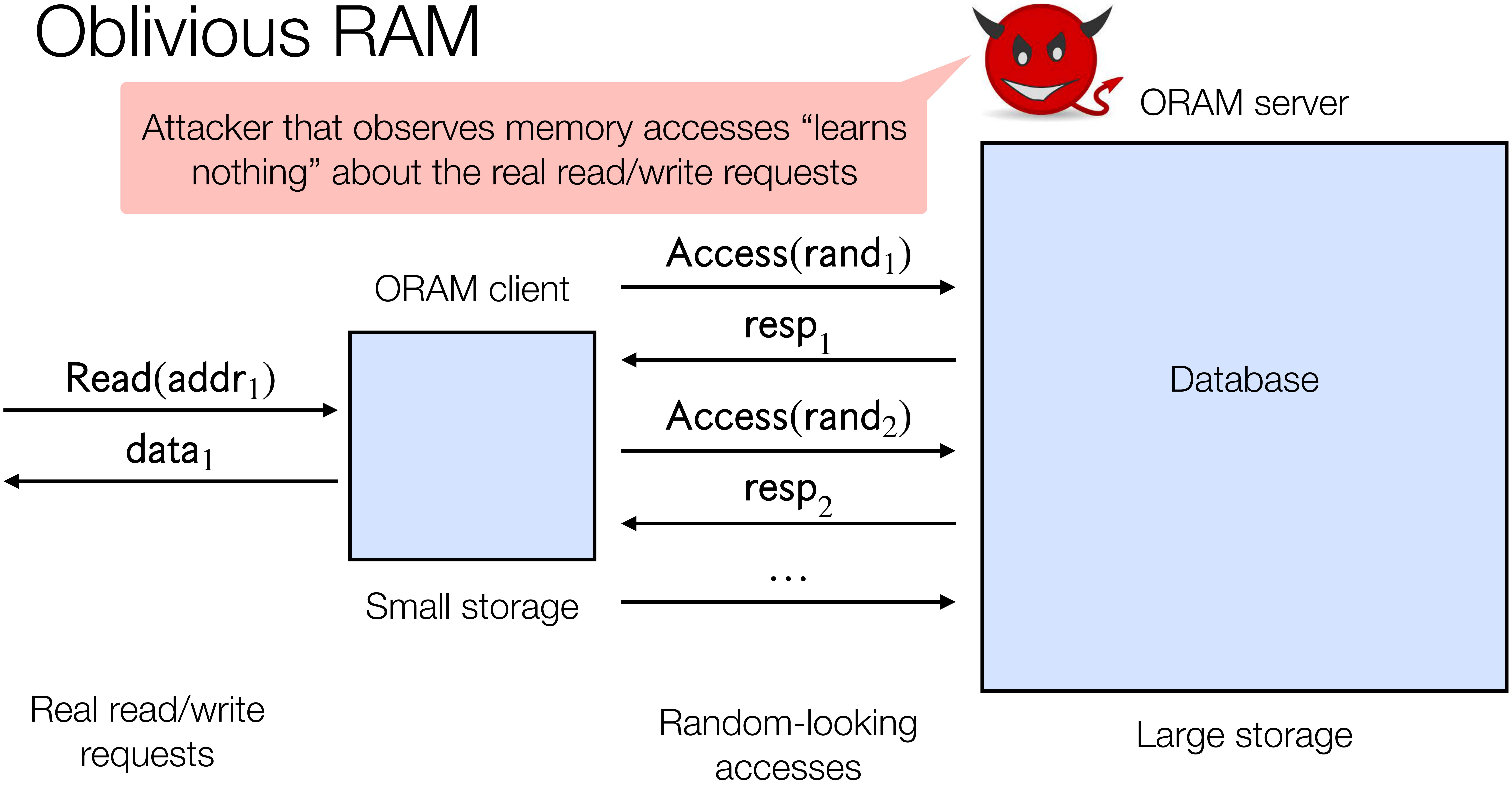
Access patterns can reveal sensitive information



Oblivious RAM



Oblivious RAM



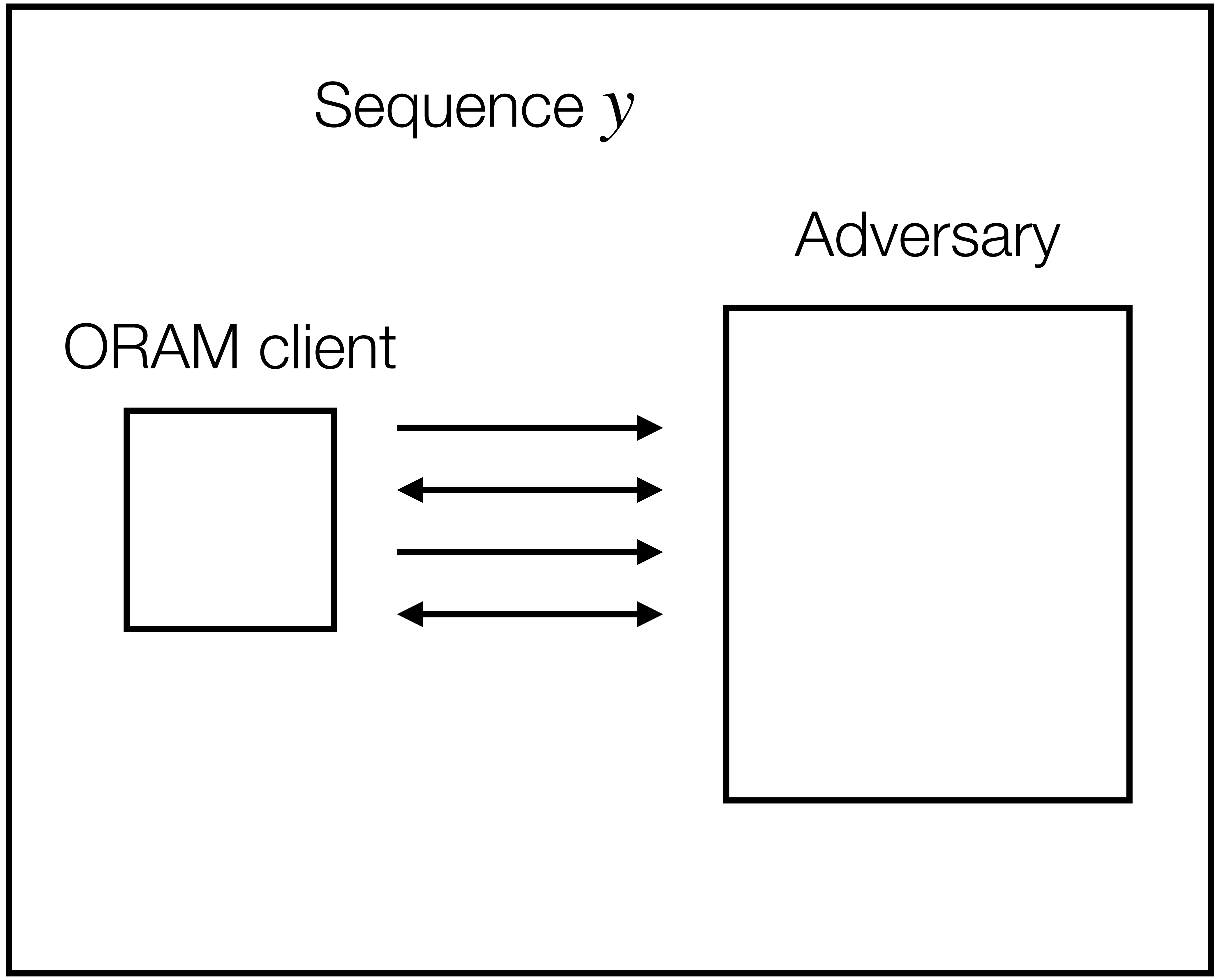
Definitions

Sequence of operations $y = ((\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1), (\mathsf{op}_2, \mathsf{addr}_2, \mathsf{data}_2), \ldots)$ where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$

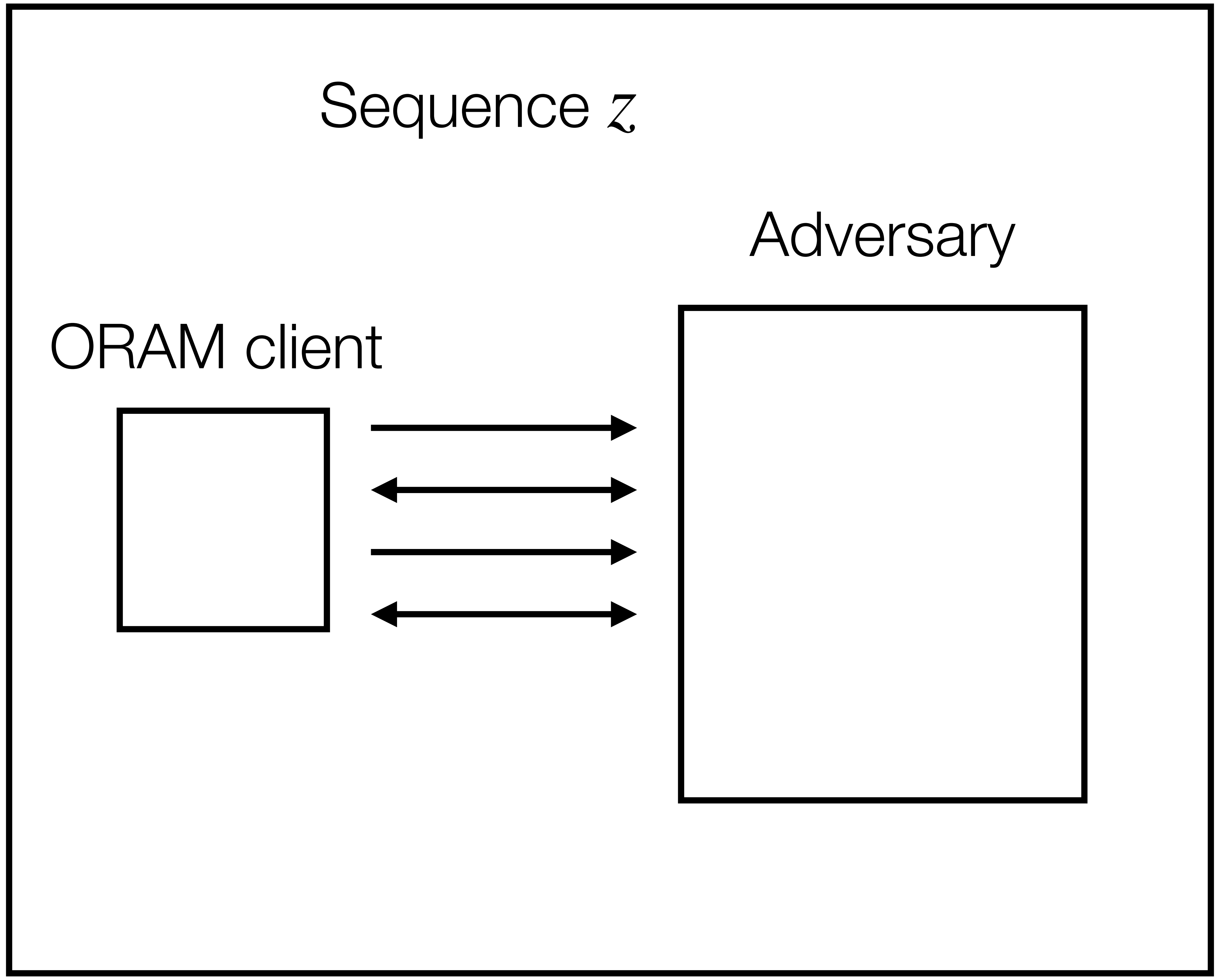
Correctness: An ORAM construction is correct if the responses from the ORAM for a sequence of operations y matches the responses from a standard RAM (with overwhelming probability).

Security: For any two request sequences y, z of the same length, their access patterns (i.e., interactions between ORAM client and server) are indistinguishable.

Security definition



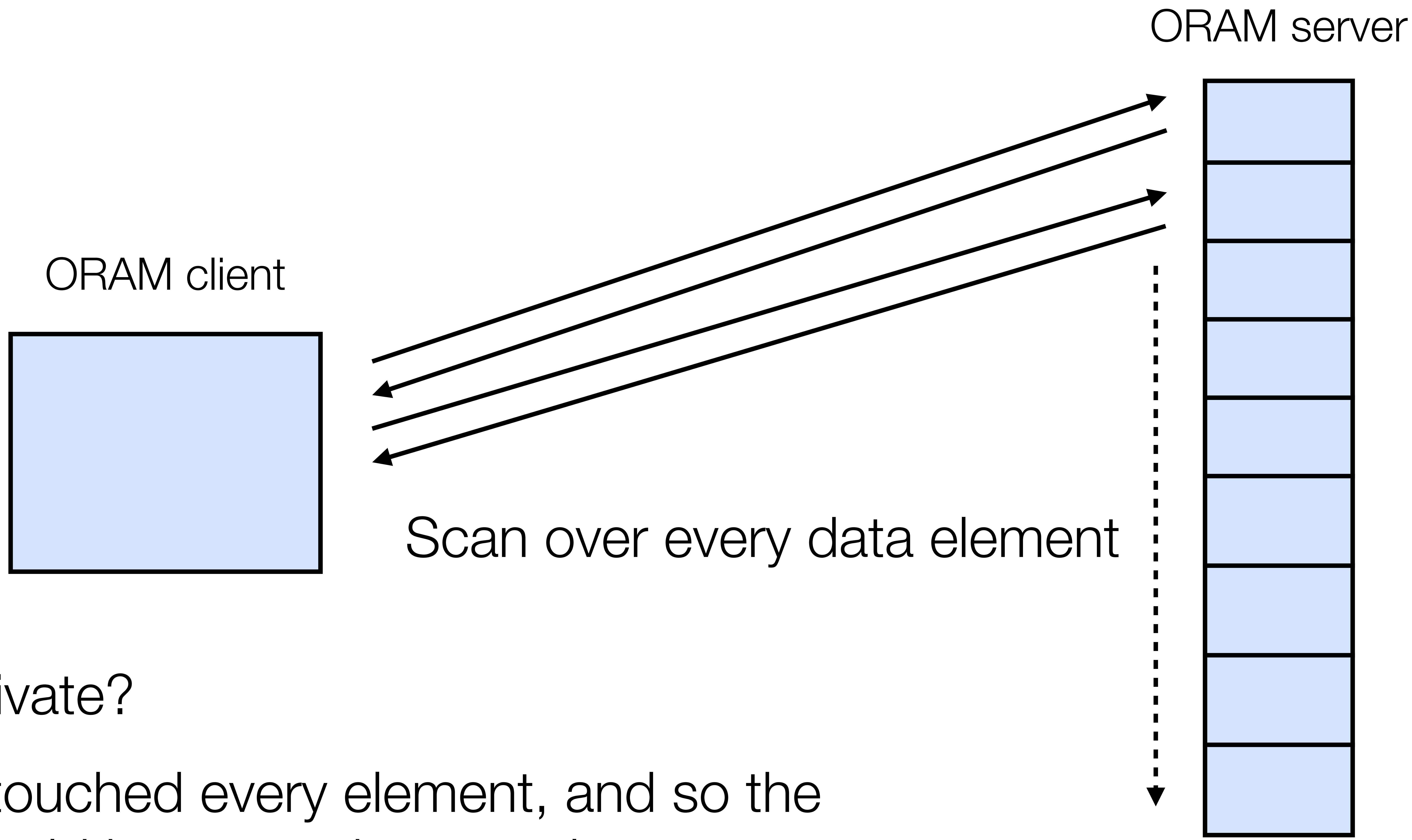
$\approx_c$



Outline

1. Overview
2. **Square-root construction**
3. Hierarchical construction
4. Limitations
5. Student presentation: PathORAM

A very simple, very expensive construction



Why private?

Server touched every element, and so the client could be accessing any element

Square-root ORAM construction

Goldreich and Ostrovsky

- Can we reduce the costs?
- Take advantage of:
  - Randomness
  - Large, mutable server state
  - Small, mutable client state

On the way to square-root ORAM construction

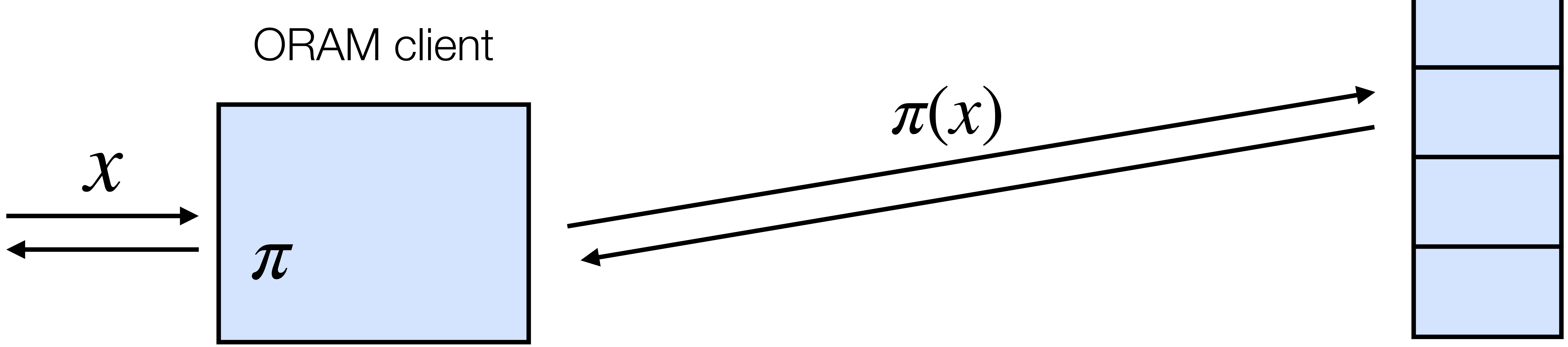
Setup: Client shuffles server data with permutation π

Access(x) :

Step 1: Client accesses address x at $\pi(x)$

Problem: Server can tell if the client is accessing the same element twice

Use oblivious sorting algorithm so that access patterns are data-independent (Batcher’s sorting)



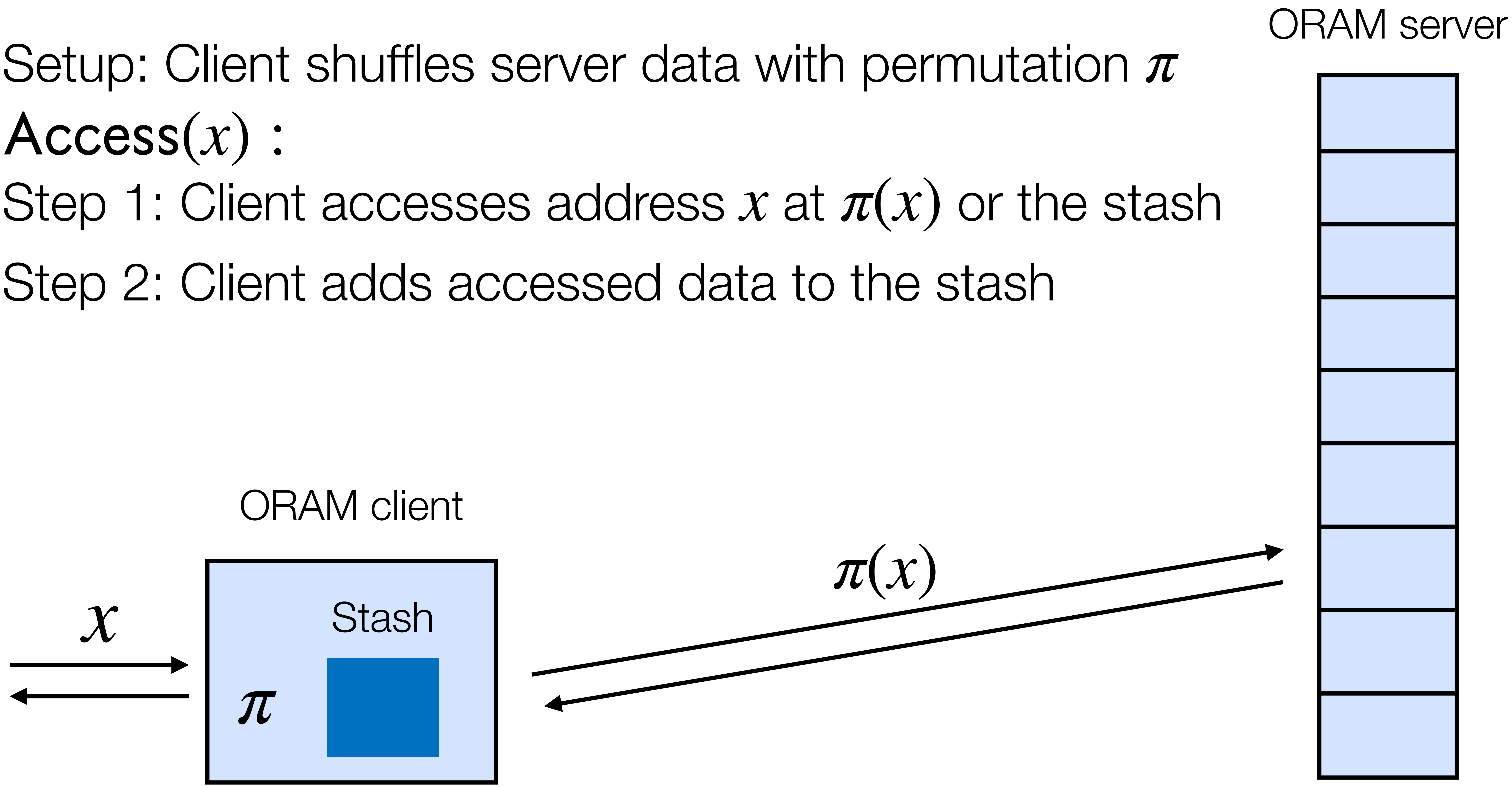
On the way to square-root ORAM construction

Setup: Client shuffles server data with permutation π

Access(x) :

Step 1: Client accesses address x at $\pi(x)$ or the stash

Step 2: Client adds accessed data to the stash



Stash stores all elements fetched with π

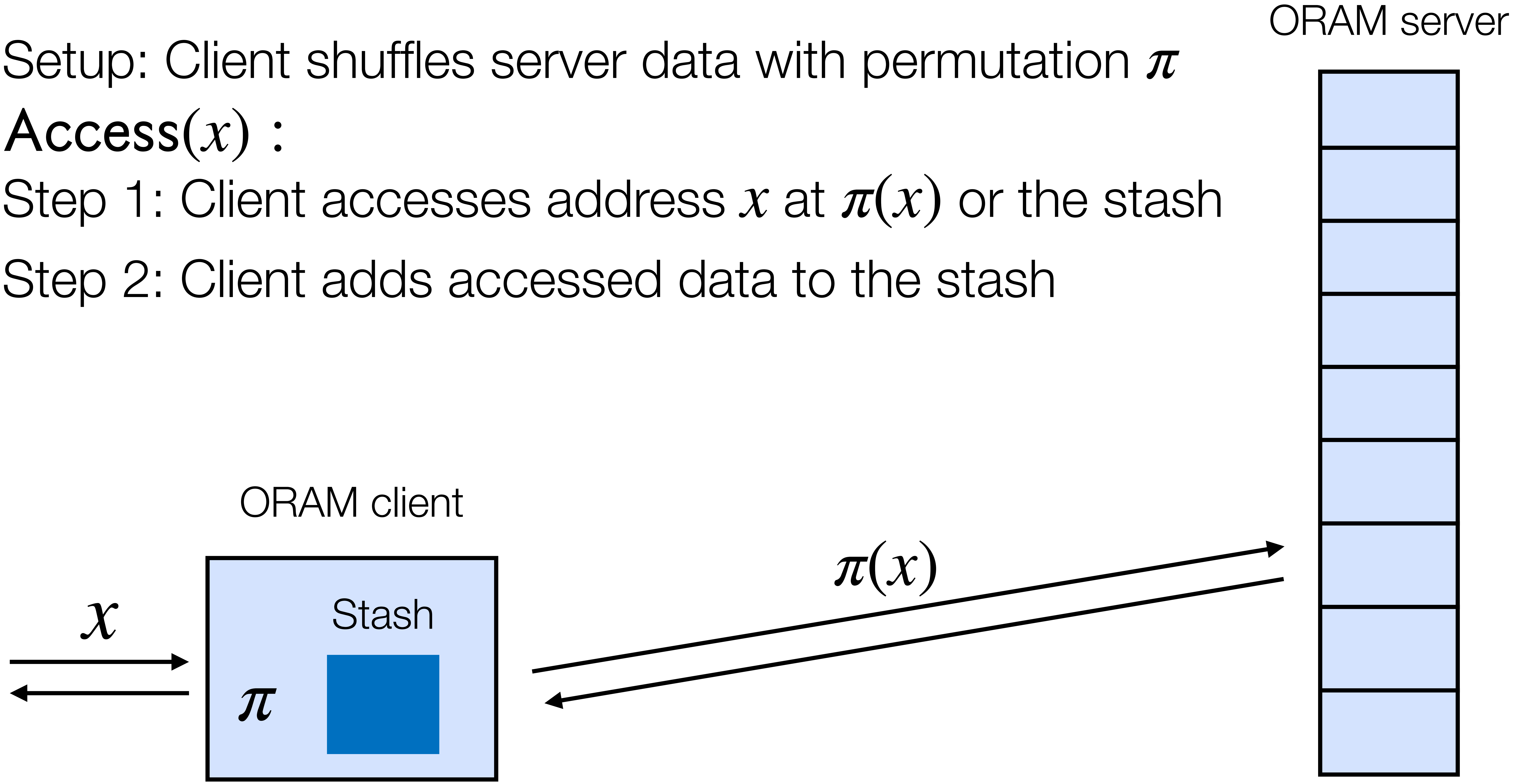
On the way to square-root ORAM construction

Setup: Client shuffles server data with permutation π

Access(x) :

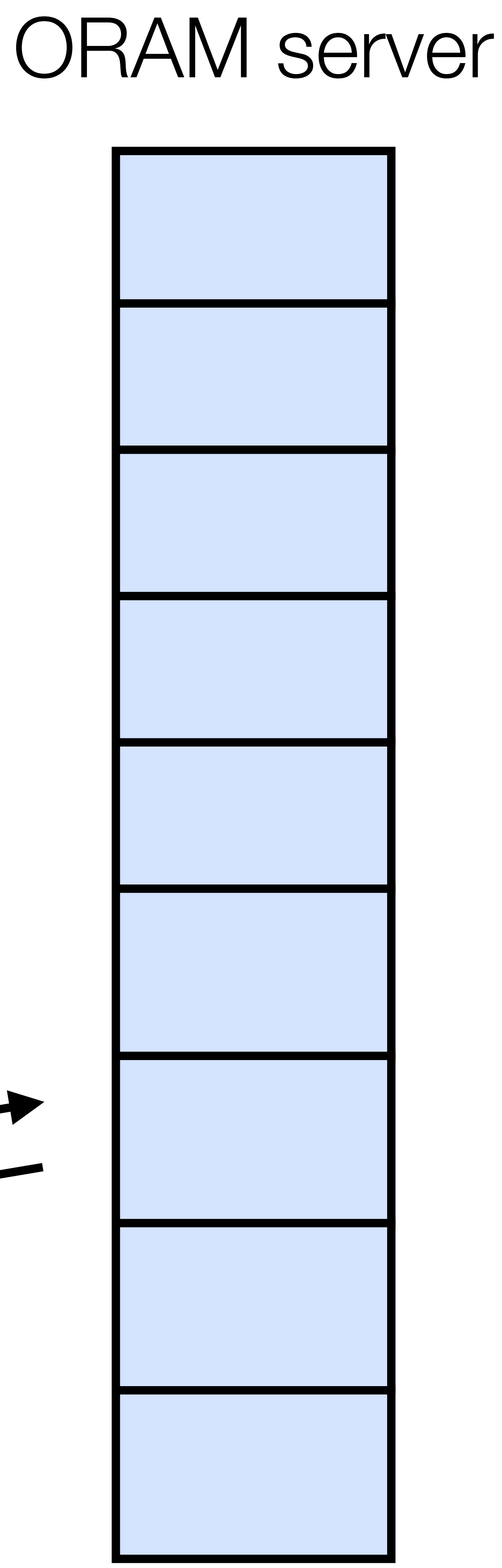
Step 1: Client accesses address x at $\pi(x)$ or the stash

Step 2: Client adds accessed data to the stash



Problem: How to keep stash from growing indefinitely?

On the way to square-root ORAM construction



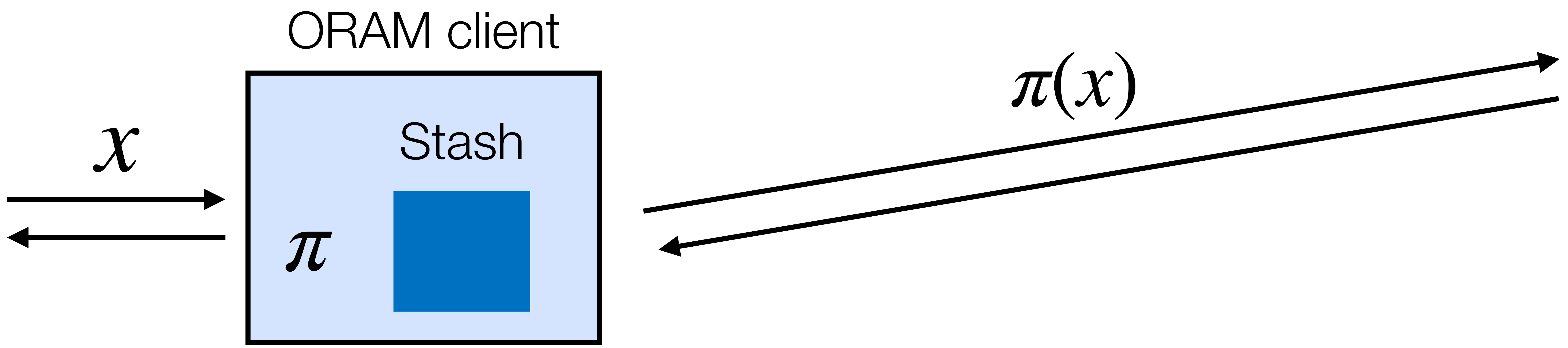
Setup: Client shuffles server data with permutation π

Access(x) :

Step 1: Client accesses address x at $\pi(x)$ or the stash

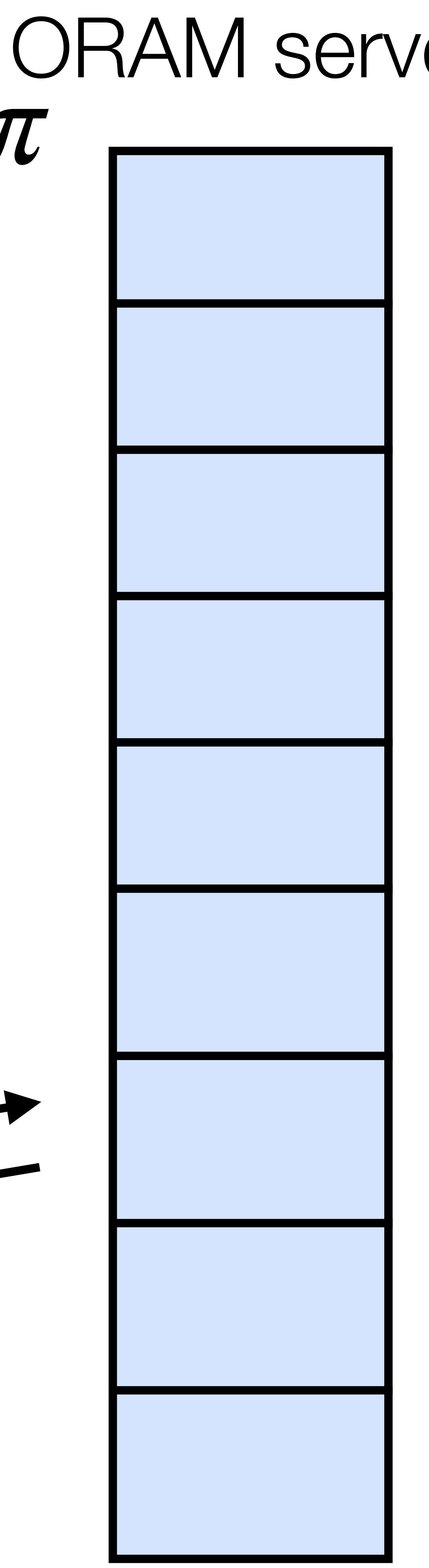
Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty



Problem: How to hide if a request is in the stash?

On the way to square-root ORAM construction



Setup: Shuffle server data + dummies with permutation π

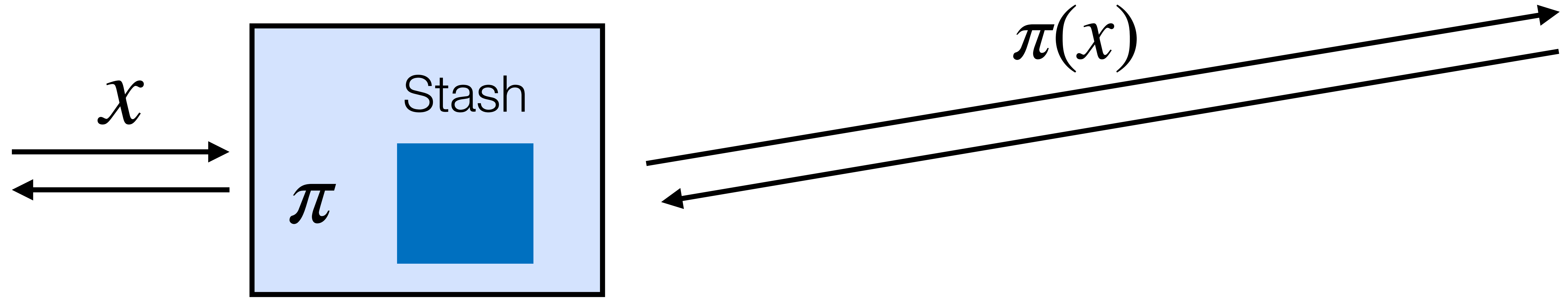
Access(x) :

Step 1: If x is in the stash, client accesses **dummy**.

Otherwise, client accesses $\pi(x)$

Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty



On the way to square-root ORAM construction

Setup: Shuffle server data + dummies with permutation π

Access(x) :

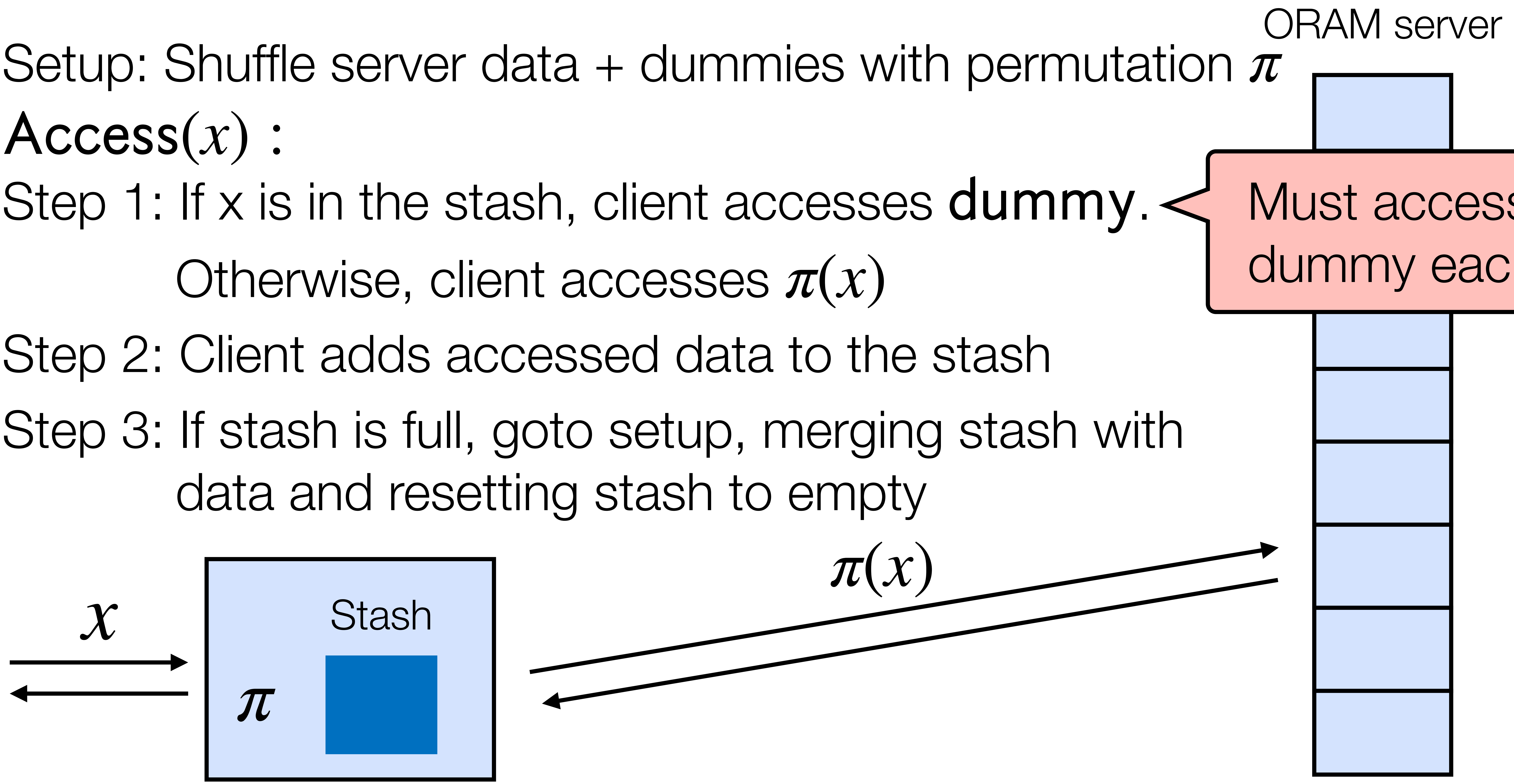
Step 1: If x is in the stash, client accesses **dummy**.

Otherwise, client accesses $\pi(x)$

Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty

Must access unique dummy each request



Correctness

Setup: Shuffle server data + dummies with permutation π

Access(x) :

Step 1: If x is in the stash, client accesses dummy.

Otherwise, client accesses $\pi(x)$

Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty

Correctness from:

- Correctness of shuffle
- Correct maintenance of the stash

Security

Setup: Shuffle server data + dummies with permutation π

Access(x) :

Step 1: If x is in the stash, client accesses **dummy**.

Otherwise, client accesses $\pi(x)$

Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty

Security from:

- Permutation appears random to the server
- Each request accesses a unique element
- Server cannot distinguish a real request from a dummy request

Parameterizing square-root ORAM

Length n array, stash size s

Add s dummies

Setup: Shuffle server data + dummies with permutation π

$O(n \cdot \log^2 n)$ comparisons

$\mathsf{Access}(x)$:

Step 1: If x is in the stash, client accesses **dummy**.

Otherwise, client accesses $\pi(x)$

Check s elements

Step 2: Client adds accessed data to the stash

Step 3: If stash is full, goto setup, merging stash with data and resetting stash to empty

By setting stash size $s = \sqrt{n}$, amortized overhead is $O(\sqrt{n} \cdot \log^2 n)$

Client storage is $O(\sqrt{n})$

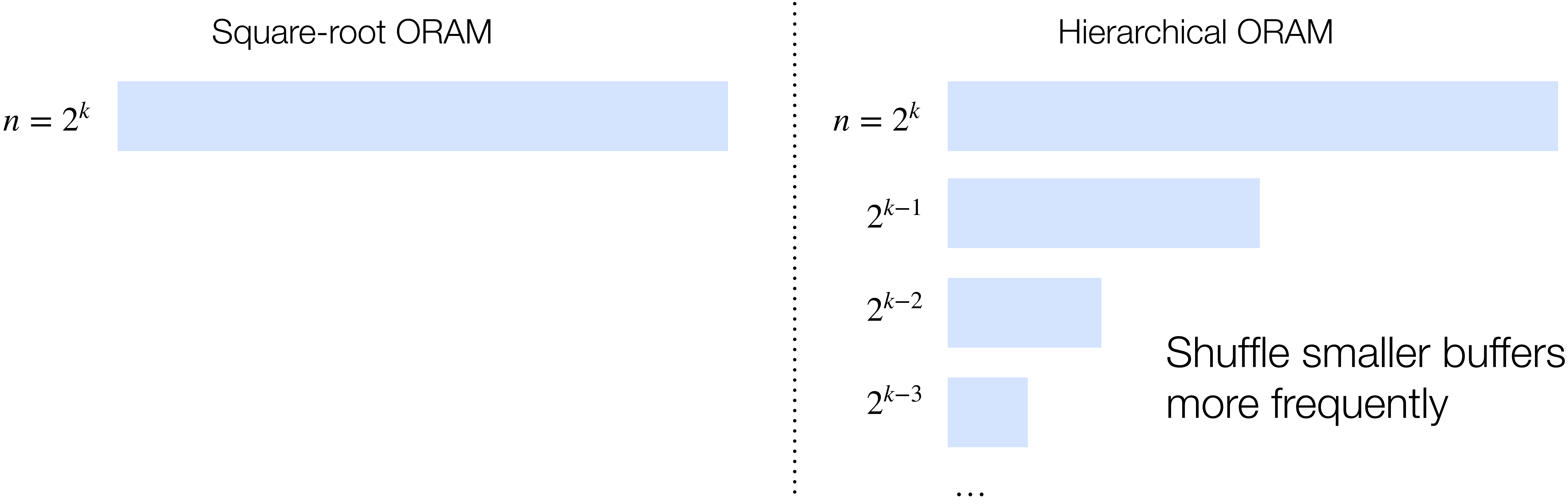
Outline

1. Overview
2. Square-root construction
3. **Hierarchical construction**
4. Limitations
5. Student presentation: PathORAM

Hierarchical ORAM

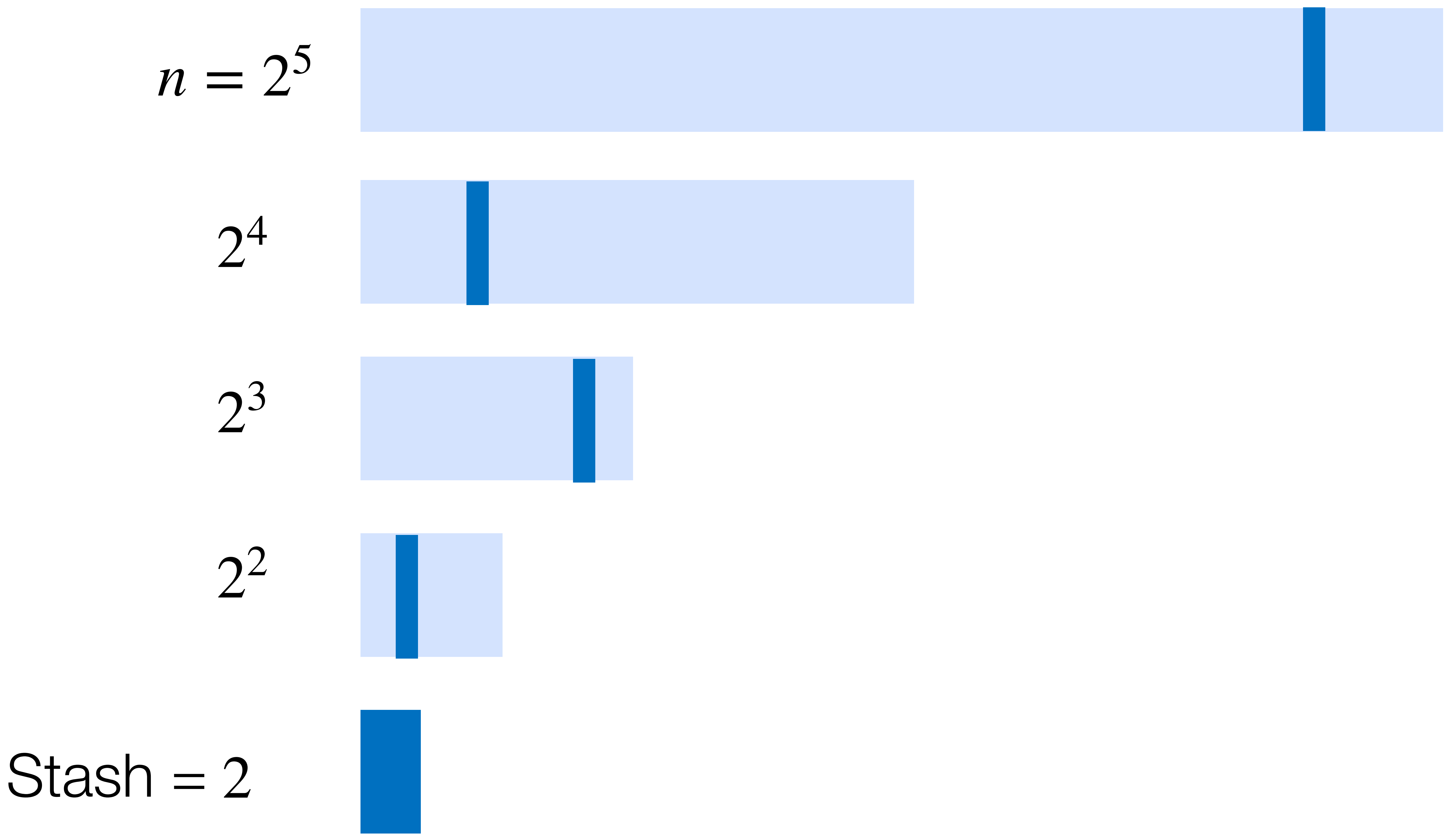
[Goldreich, Ostrovsky]

High-level idea: Hierarchy of different-sized buffers to achieve logarithmic overhead



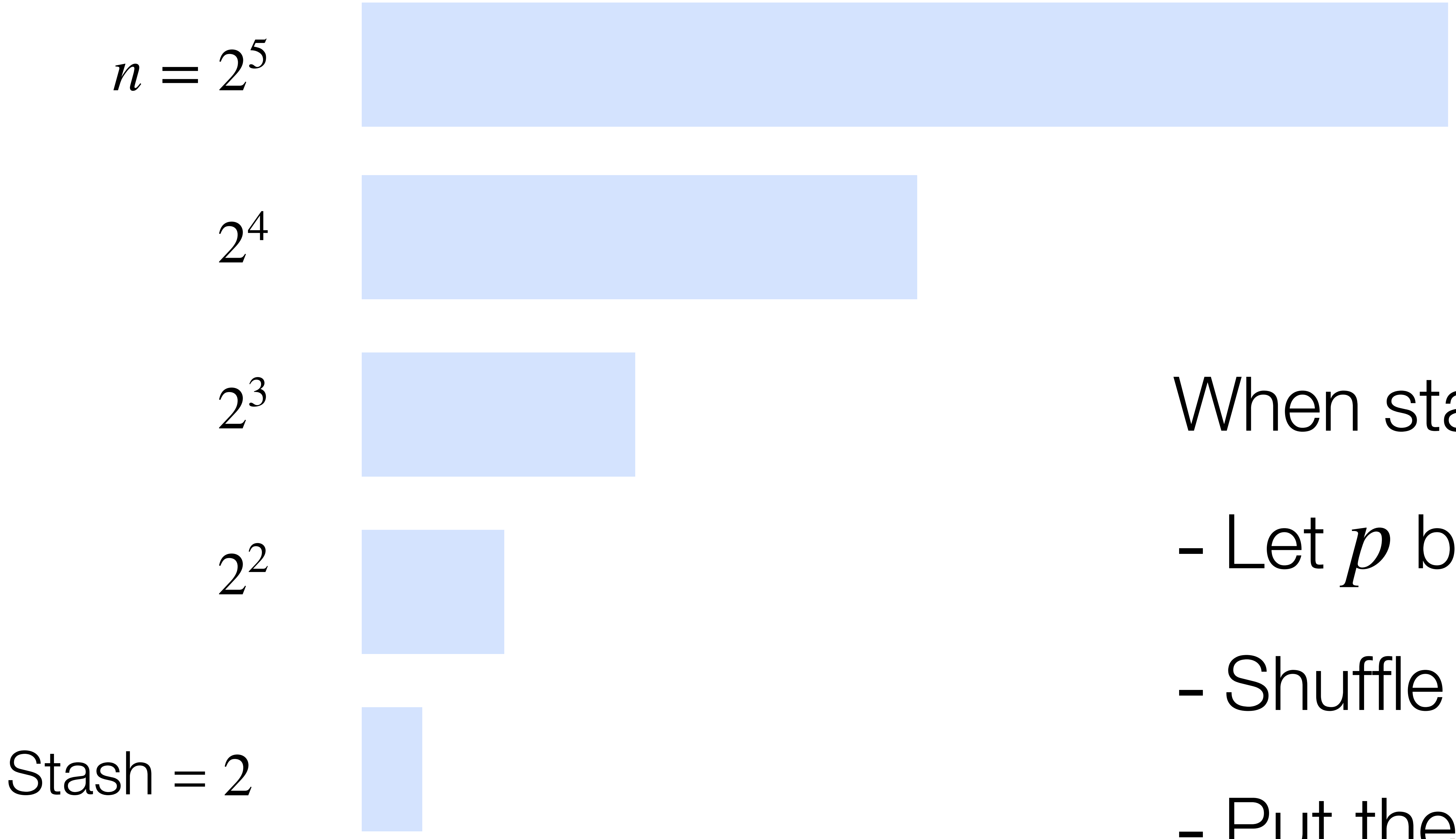
Hierarchical ORAM

Access requires scanning over the stash and making a lookup in each buffer



Hierarchical ORAM

Shuffle smaller buffers more frequently

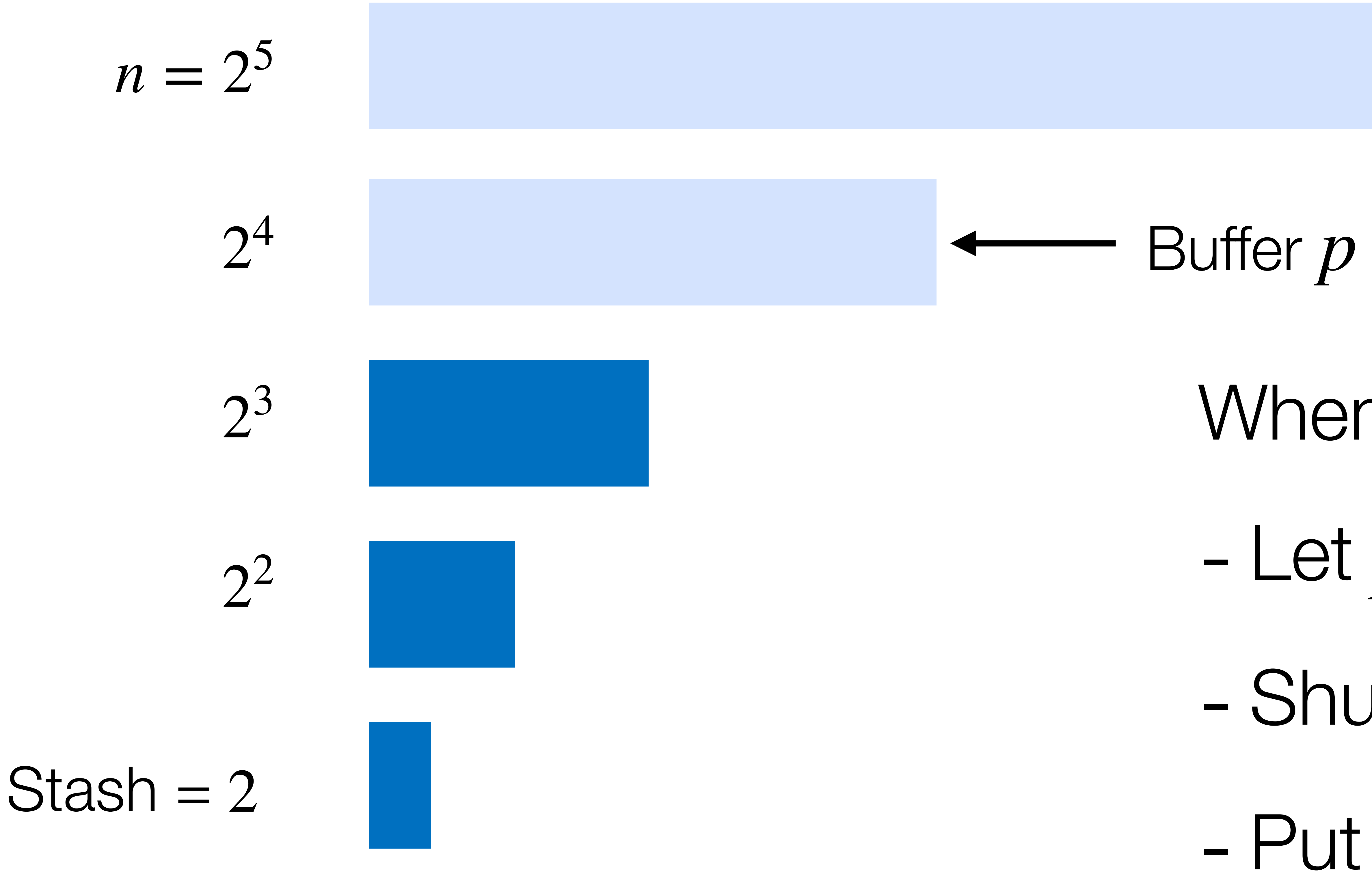


When stash is full:

- Let p be the index of the smallest unfilled buffer
- Shuffle together the stash and buffers $i < p$
- Put the results in buffer p

Hierarchical ORAM

Shuffle smaller buffers more frequently

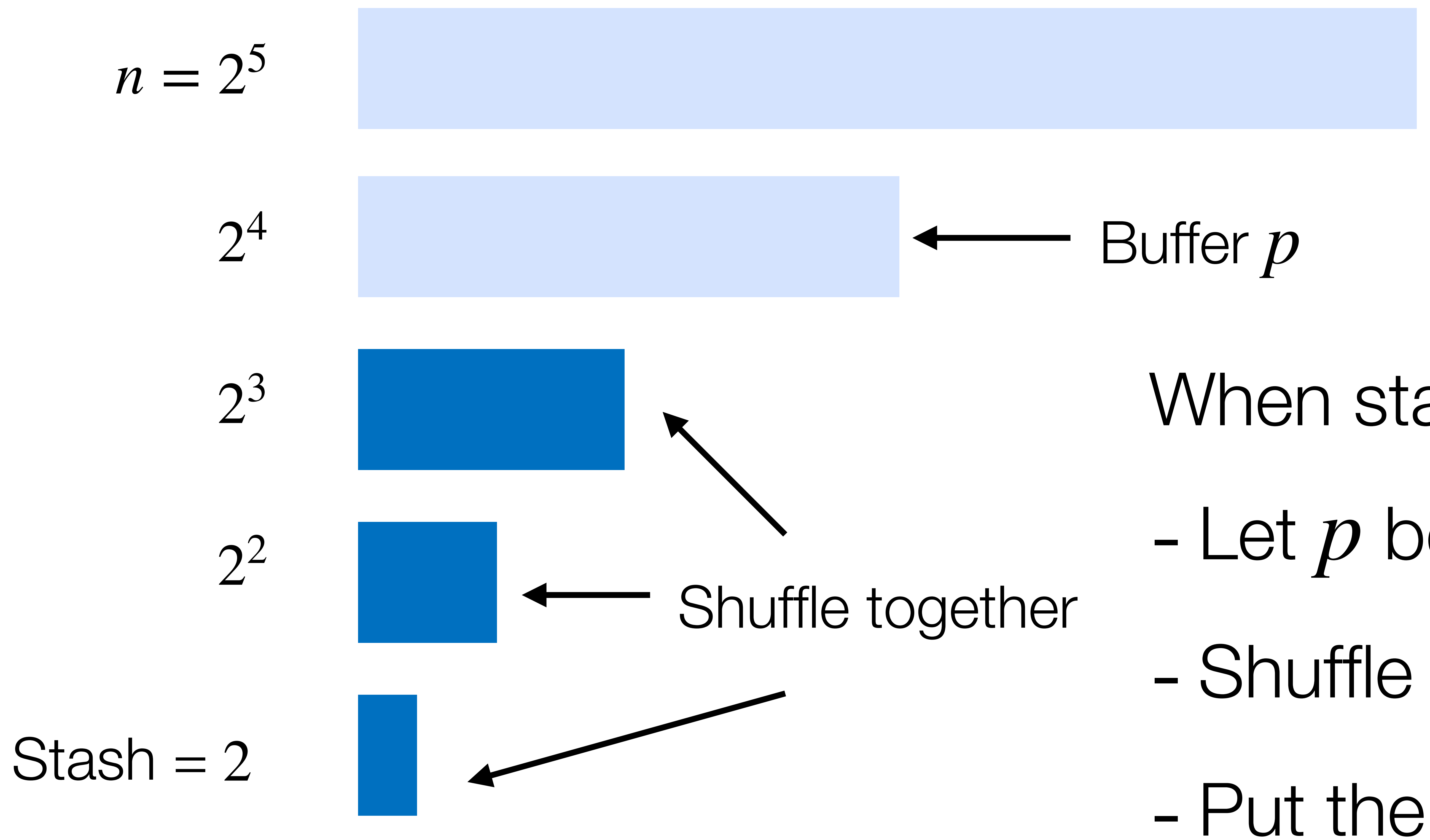


When stash is full:

- Let p be the index of the smallest unfilled buffer
- Shuffle together the stash and buffers $i < p$
- Put the results in buffer p

Hierarchical ORAM

Shuffle smaller buffers more frequently

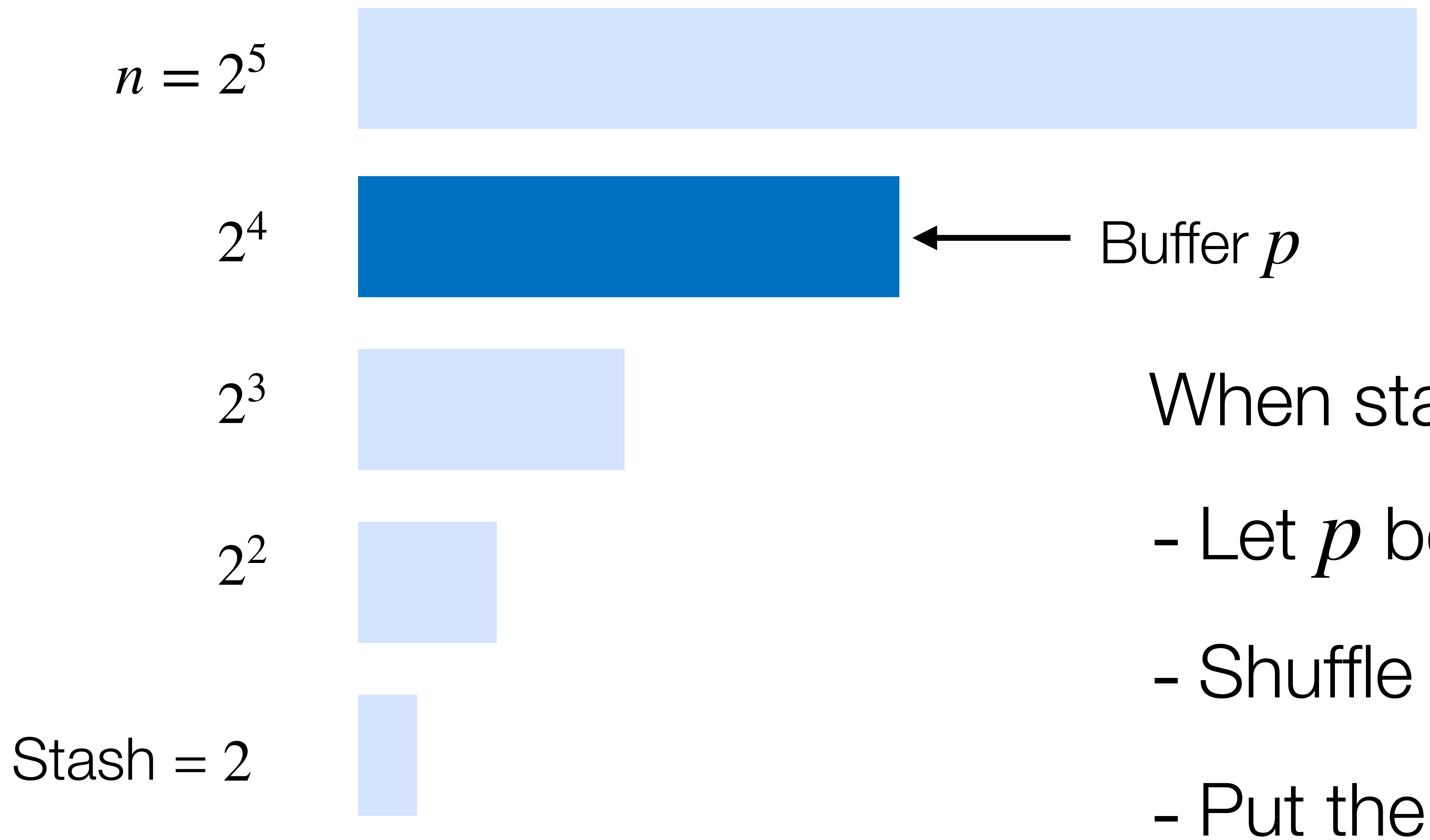


When stash is full:

- Let p be the index of the smallest unfilled buffer
- Shuffle together the stash and buffers $i < p$
- Put the results in buffer p

Hierarchical ORAM

Shuffle smaller buffers more frequently



When stash is full:

- Let p be the index of the smallest unfilled buffer
- Shuffle together the stash and buffers $i < p$
- Put the results in buffer p

Outline

1. Overview
2. Square-root construction
3. Hierarchical construction
4. **Limitations**
5. Student presentation: PathORAM

Limitations of ORAM

- All elements must be the same length
- Increased cost compared to plaintext RAM accesses: lower bound of $O(\log n)$ accesses per operation [Larsen, Nielsen]
- ORAM is designed for a single client; does not directly support multiple clients
- Accesses where shuffling is required take longer than accesses without shuffling (not the case for tree-based ORAMs)
- Elements must be fetched in sequence, not in parallel (addressed in subsequent work, e.g.,TaoStore)
- Only supports key-value lookups, but applications need other types of queries

Outline

1. Overview
2. Square-root construction
3. Hierarchical construction
4. Limitations
5. **Student presentation: PathORAM**

References

Goldreich, Oded, and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs." *Journal of the ACM (JACM)* 43.3 (1996): 431-473.

Larsen, Kasper Green, and Jesper Buus Nielsen. "Yes, there is an oblivious RAM lower bound!." *Annual International Cryptology Conference*. Cham: Springer International Publishing, 2018.

Sahin, Cetin, et al. "Taostore: Overcoming asynchronicity in oblivious data storage." *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

Stefanov, Emil, et al. "Path ORAM: an extremely simple oblivious RAM protocol." *Journal of the ACM (JACM)* 65.4 (2018): 1-26

https://6893.csail.mit.edu/lec7.pdf