# CS 350S: Privacy-Preserving Systems

## Oblivious RAM II

# Outline

1. Ring ORAM
2. Transactions
3. Obladi
4. Horizontally scalable ORAM
5. Logistics

# Recap: PathORAM

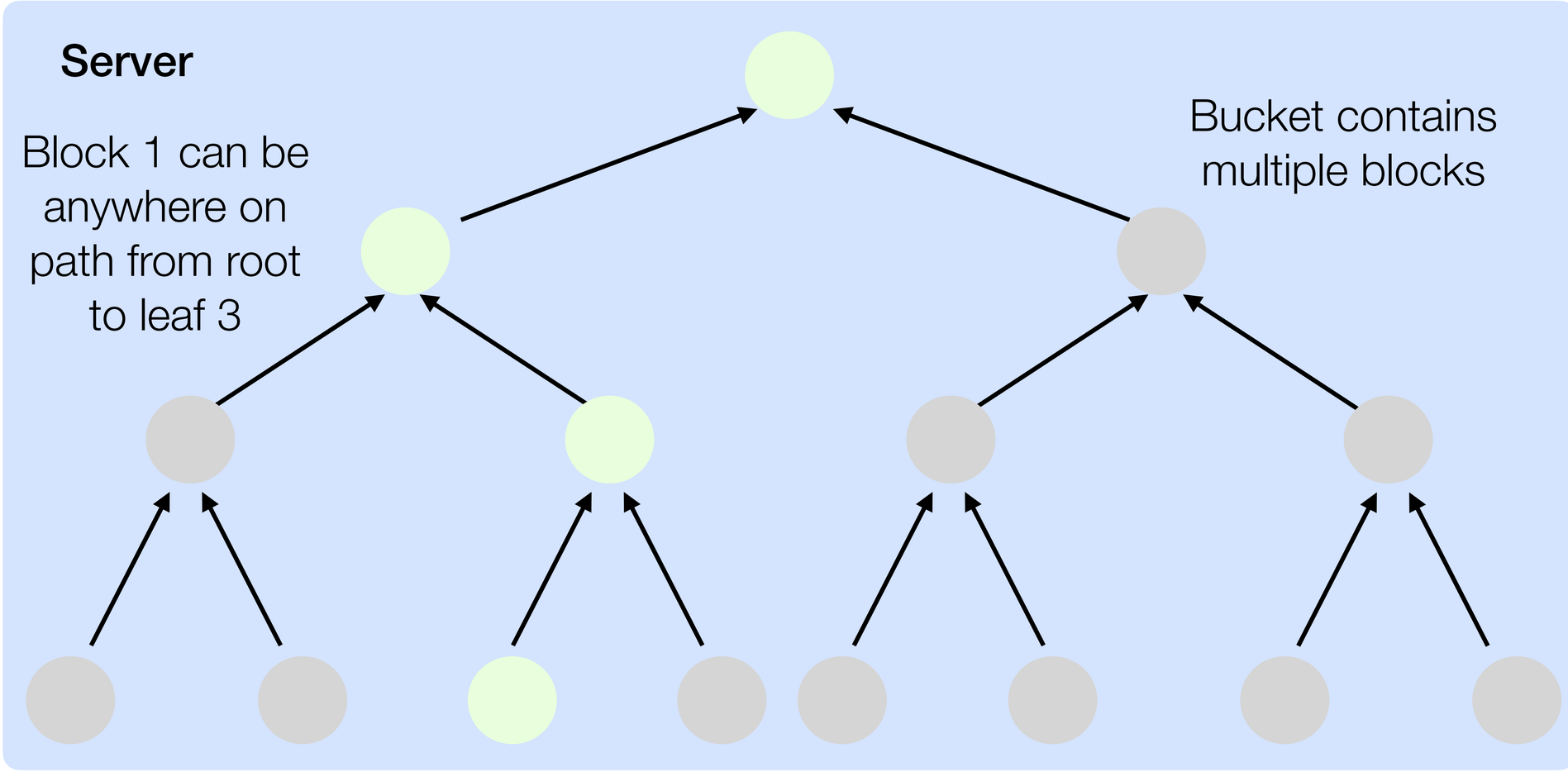
[Stefanov, Van Dijk, Shi, Chan, Fletcher, Ren, Yu, Devadas]

Tree where blocks are assigned to a path

**Client**

Position map:
- block 1: path 3
- …

Stash



# Recap: PathORAM

1. Read the path storing the address a, put the block in the stash, and choose a new random path for it
2. Write back the path just read with elements from the stash

| | |
|---|---|
| $N$ | Total # blocks outsourced to server |
| $L$ | Height of binary tree |
| $B$ | Block size (in bits) |
| $Z$ | Capacity of each bucket (in blocks) |
| $\mathcal{P}(x)$ | Path from leaf node $x$ to the root |
| $\mathcal{P}(x, \ell)$ | The bucket at level $\ell$ along the path $\mathcal{P}(x)$ |
| $S$ | Client's local stash |
| position | Client's local position map |

Access(op, a, data$^*$):

1: $x \leftarrow$ position[a]
2: position[a] $\leftarrow x^* \leftarrow$ UniformRandom$(0 \ldots 2^L - 1)$
3: **for** $\ell \in \{0, 1, \ldots, L\}$ **do**
4: $\quad S \leftarrow S \cup$ ReadBucket$(\mathcal{P}(x, \ell))$
5: **end for**
6: data $\leftarrow$ **Read block** a **from** $S$
7: **if** op $=$ write **then**
8: $\quad S \leftarrow (S - \{(\mathsf{a}, x, \mathsf{data})\}) \cup \{(\mathsf{a}, x^*, \mathsf{data}^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, \ldots, 0\}$ **do**
11: $\quad S' \leftarrow \{(\mathsf{a}', x', \mathsf{data}') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(x', \ell)\}$
12: $\quad S' \leftarrow$ **Select** $\min(|S'|, Z)$ **blocks from** $S'$.
13: $\quad S \leftarrow S - S'$
14: $\quad$ WriteBucket$(\mathcal{P}(x, \ell), S')$
15: **end for**
16: **return** data

# Ring ORAM

[Ren, Fletcher, Kwon, Stefanov, Shi, van Dijk, Devadas]

Reduces Path ORAM bandwidth: communication independent of bucket size

Invariants for security:

1. Every block is mapped to a randomly chosen path (from Path ORAM)
2. Blocks in each bucket are randomly permuted with respect to past and future writes (new)

**Algorithm 1** Non-recursive Ring ORAM.

1: **function** ACCESS($a$, op, data$'$)
2: **Global/persistent variables:** round
3: $l' \leftarrow$ UniformRandom$(0, 2^L - 1)$
4: $l \leftarrow$ PositionMap$[a]$
5: PositionMap$[a] \leftarrow l'$
6: data $\leftarrow$ ReadPath$(l, a)$
7: **if** data $= \perp$ **then**
8: $\triangleright$ If block $a$ is not found on path $l$, it must
9: be in Stash $\triangleleft$
10: data $\leftarrow$ read and remove $a$ from Stash
11: **if** op $=$ read **then**
12: return data to client
13: **if** op $=$ write **then**
14: data $\leftarrow$ data$'$
15: Stash $\leftarrow$ Stash $\cup (a, l', \text{data})$
16: round $\leftarrow$ round $+ 1 \mod A$
17: **if** round $\stackrel{?}{=} 0$ **then**
18: EvictPath()
19: EarlyReshuffle$(l)$

**Algorithm 2** ReadPath procedure.

```
1: function ReadPath(l, a)
2:     data ← ⊥
3:     for i ← 0 to L do
4:         offset ← GetBlockOffset(P(l, i), a)
5:         data′ ← P(l, i, offset)
6:         Invalidate P(l, i, offset)
7:         if data′ ≠ ⊥ then
8:             data ← data′
9:         P(l, i).count ← P(l, i).count + 1
       return data
```

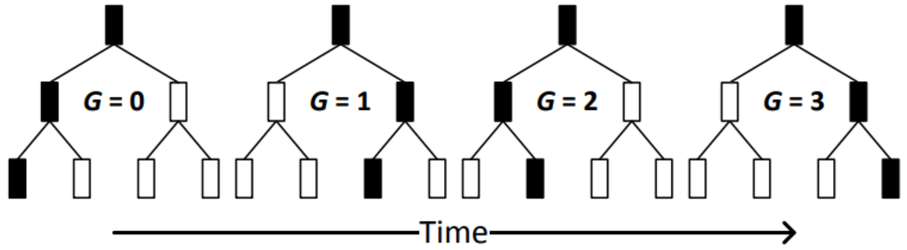
Fetch metadata for block (much smaller than actual data)

Then fetch either dummy or real block

Can send back 1 block by XOR'ing L blocks together (1 real, L-1 dummies)

**Algorithm 1** Non-recursive Ring ORAM.

```
1: function ACCESS(a, op, data′)
2:     Global/persistent variables: round
3:     l′ ← UniformRandom(0, 2^L − 1)
4:     l ← PositionMap[a]
5:     PositionMap[a] ← l′
6:     data ← ReadPath(l, a)
7:     if data = ⊥ then
8:         ▷ If block a is not found on path l, it must
9:         be in Stash ◁
10:        data ← read and remove a from Stash
11:    if op = read then
12:        return data to client
13:    if op = write then
14:        data ← data′
15:    Stash ← Stash ∪ (a, l′, data)
16:    round ← round + 1  mod A
17:    if round =? 0 then
18:        EvictPath()
19:    EarlyReshuffle(l)
```



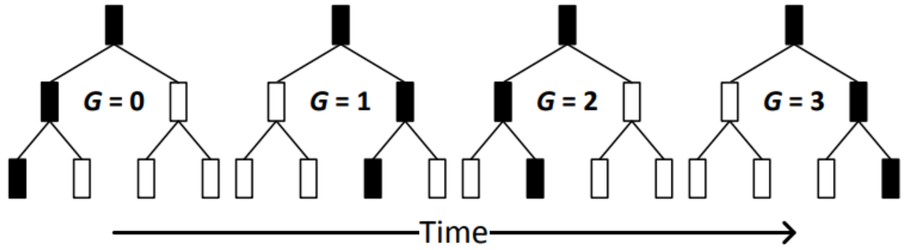

Eviction: read blocks in along a path and push stash blocks as far along the path as possible

Eviction in reverse-lexicographic order to maximize spread across tree

Evictions are now in a deterministic order

**Algorithm 1** Non-recursive Ring ORAM.

```
1:  function ACCESS(a, op, data′)
2:      Global/persistent variables: round
3:      l′ ← UniformRandom(0, 2^L − 1)
4:      l ← PositionMap[a]
5:      PositionMap[a] ← l′
6:      data ← ReadPath(l, a)
7:      if data = ⊥ then
8:          ▷ If block a is not found on path l, it must
9:          be in Stash ◁
10:         data ← read and remove a from Stash
11:     if op = read then
12:         return data to client
13:     if op = write then
14:         data ← data′
15:     Stash ← Stash ∪ (a, l′, data)
16:     round ← round + 1  mod A
17:     if round ≟ 0 then
18:         EvictPath()
19:     EarlyReshuffle(l)
```




**Algorithm 3** EvictPath procedure.

```
1: function EvictPath
2:     Global/persistent variables G initialized to 0
3:     l ← G mod 2^L
4:     G ← G + 1
5:     for i ← 0 to L do
6:         Stash ← Stash ∪ ReadBucket(P(l, i))
7:     for i ← L to 0 do
8:         WriteBucket(P(l, i), Stash)
9:         P(l, i).count ← 0
```

**Algorithm 1** Non-recursive Ring ORAM.

```
1: function ACCESS(a, op, data′)
2:     Global/persistent variables: round
3:     l′ ← UniformRandom(0, 2^L − 1)
4:     l ← PositionMap[a]
5:     PositionMap[a] ← l′
6:     data ← ReadPath(l, a)
7:     if data = ⊥ then
8:         ▷ If block a is not found on path l, it must
9:         be in Stash ◁
10:        data ← read and remove a from Stash
11:    if op = read then
12:        return data to client
13:    if op = write then
14:        data ← data′
15:    Stash ← Stash ∪ (a, l′, data)
16:    round ← round + 1 mod A
17:    if round =? 0 then
18:        EvictPath()
19:    EarlyReshuffle(l)
```

**Algorithm 4** EarlyReshuffle procedure.

```
1: function EarlyReshuffle(l)
2:     for i ← 0 to L do
3:         if P(l,i).count ≥ S then
4:             Stash ← Stash ∪ ReadBucket(P(l,i))
5:             WriteBucket(P(l,i), Stash)
6:             P(l,i).count ← 0
```

If a bucket on the read path has been accessed more than S times, reshuffle the blocks in that bucket

**Algorithm 1** Non-recursive Ring ORAM.

```
1: function ACCESS(a, op, data′)
2:     Global/persistent variables: round
3:     l′ ← UniformRandom(0, 2^L − 1)
4:     l ← PositionMap[a]
5:     PositionMap[a] ← l′
6:     data ← ReadPath(l, a)
7:     if data = ⊥ then
8:         ▷ If block a is not found on path l, it must
9:         be in Stash ◁
10:        data ← read and remove a from Stash
11:    if op = read then
12:        return data to client
13:    if op = write then
14:        data ← data′
15:    Stash ← Stash ∪ (a, l′, data)
16:    round ← round + 1  mod A
17:    if round =? 0 then
18:        EvictPath()
19:    EarlyReshuffle(l)
```

# Outline

1. Ring ORAM
2. **Transactions**
3. Obladi
4. Horizontally scalable ORAM
5. Logistics

# Transactions

Transaction: set of operations that are treated as a single unit

**A**tomicity: either all of the operations in the transaction are executed, or none are

**C**onsistency: multiple parties access shared data according to predefined rules (i.e., the consistency model)

**I**solation: ensures that concurrent transactions don't interfere with each other (illusion that each transaction is running one-at-a-time)

**D**urability: the effects of successful transactions are recorded, even in the event of failure

# Transactions

Example: bank transfer where Alice sends $500 to Bob

1. Deduct $500 from Alice's account
2. Add $500 to Bob's account

Don't want $500 to be deducted from Alice's account, but Bob never receives it

# Outline

1. Ring ORAM
2. Transactions
3. **Obladi**
4. Horizontally scalable ORAM
5. Logistics

# Motivation

What we know how to do with strong cryptographic privacy

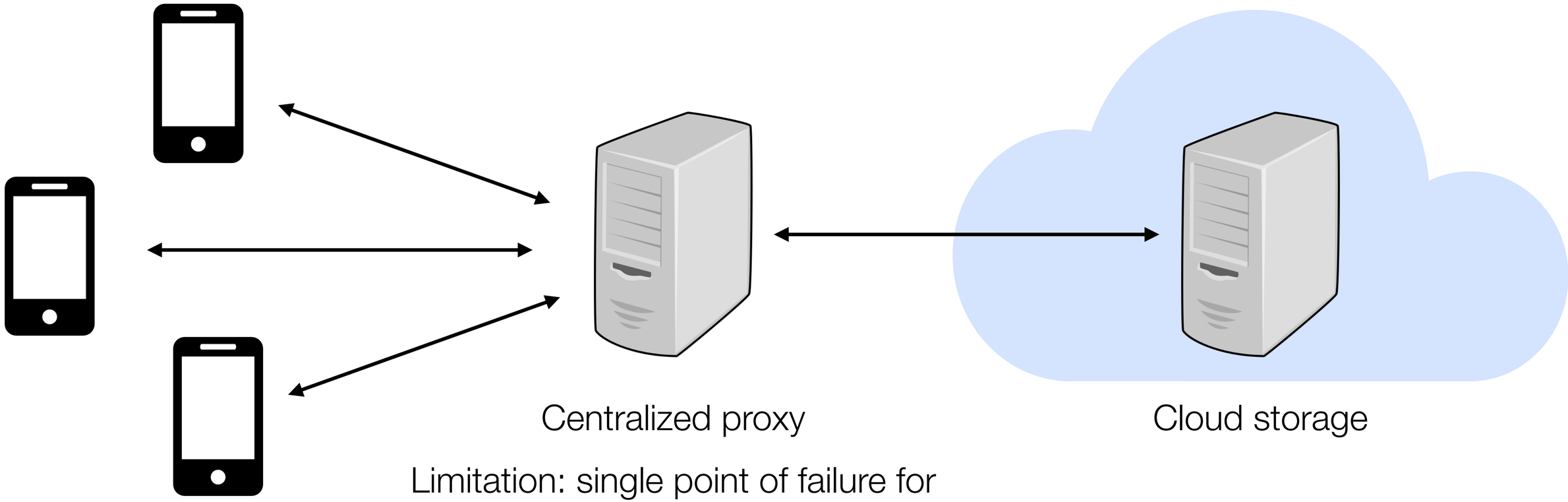
Oblivious block store

Obladi

What we need to build real-world systems

Transactions with ACID semantics (atomicity, consistency, isolation, durability)

# Obladi system architecture



Clients

Centralized proxy

Limitation: single point of failure for privacy, fault-tolerance, and durability

Cloud storage

# Obladi workload independence

Workload independence

- Sequence of Obladi transactions, where each transaction contains $(\mathsf{op}_1, \mathsf{addr}_1, \mathsf{val}_1), \ldots, (\mathsf{op}_n, \mathsf{addr}_n, \mathsf{val}_n)$
- For any two sequences of Obladi transactions, an adversary that controls the cloud storage cannot distinguish between the set of access patterns to cloud storage.
- Note: hides whether a transaction committed or aborted, the type, number, and access set of transactions, and the data values in a transaction

# Straw man design

- Keep a queue of incoming transactions and, for each transaction, process the operations one-at-a-time

Does this work?

- Secure and correctly orders transactions
- Slow: doesn’t exploit parallelism
- Lose data if there is a crash

# Obladi’s approach

Support concurrent transactions:

- How to maintain consistency while running transactions simultaneously?
- How to exploit parallelism for performance?

Durability: Ensure committed transactions are recorded even if there’s a crash

# Obladi’s approach

Support concurrent transactions:

- **How to maintain consistency while running transactions simultaneously?**
- How to exploit parallelism for performance?

Durability: Ensure committed transactions are recorded even if there’s a crash

# Concurrency requirements

Performance benefits of executing transactions concurrently

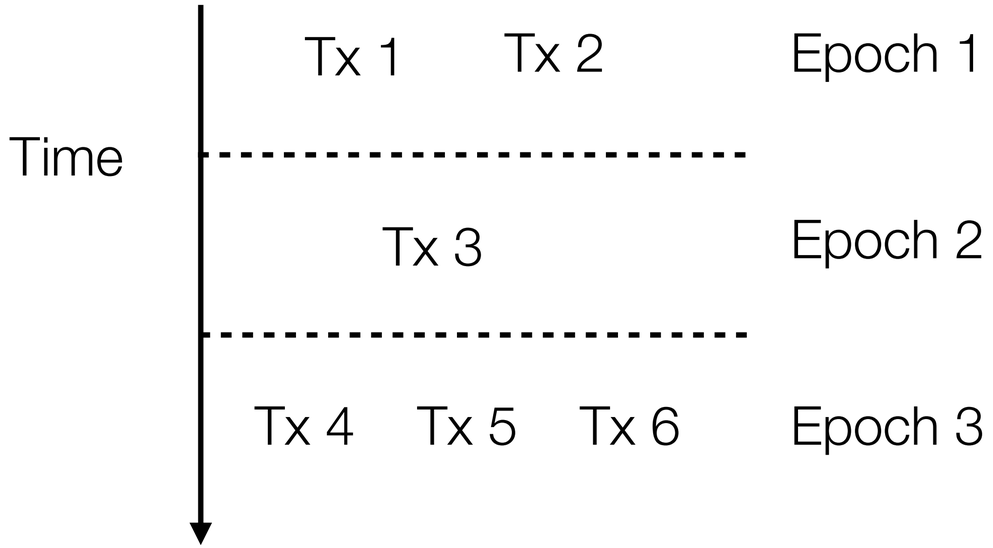
- Parallelism when servicing requests with minimal blocking

Appearance of executing requests sequentially

- Serializability: concurrent execution should appear the same as executing transactions one-at-a-time

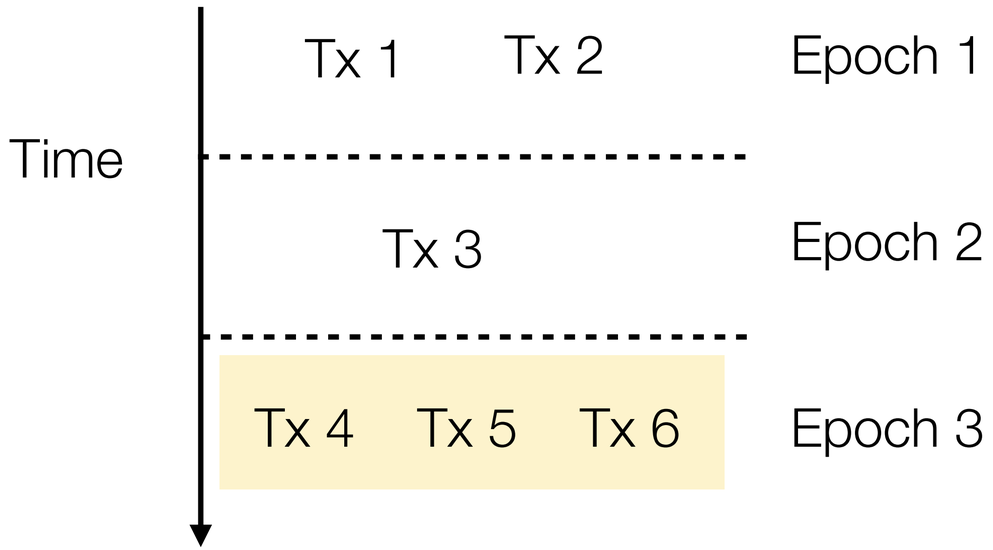
# Step 1: Split transactions across epochs

- Ordering at a coarse granularity: split time into fixed-length epochs
- Each transaction assigned to an epoch (transactions cannot span epochs)
- Transactions only commit and are made durable at the end of an epoch



# Step 2: Execute transactions within an epoch

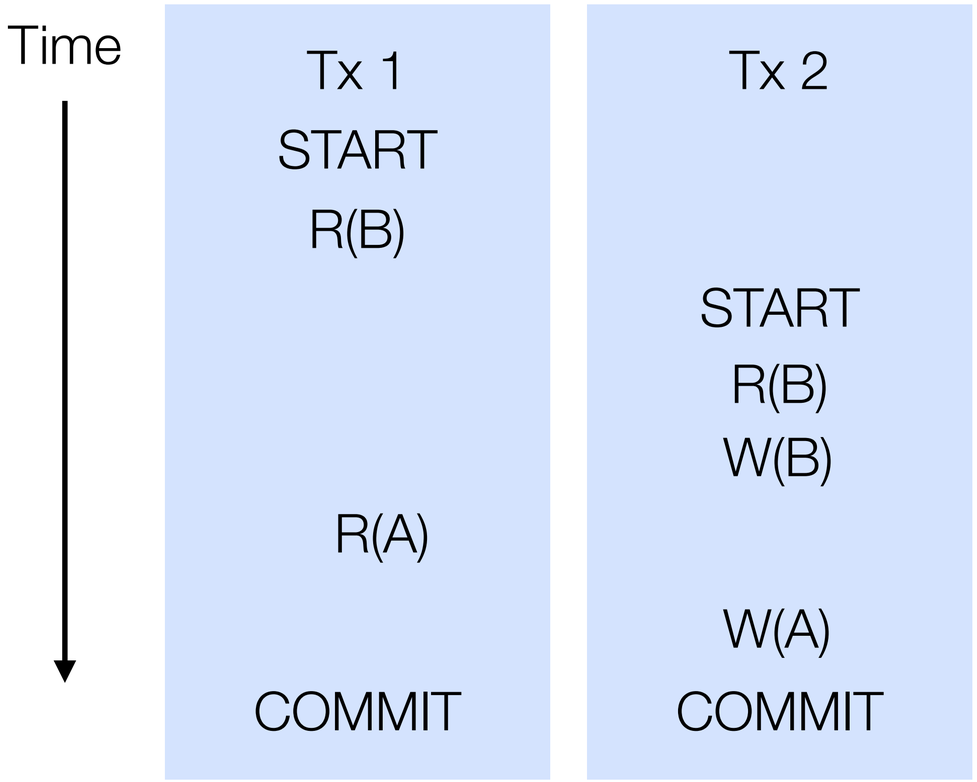
- Uncommitted writes can be observed by concurrent transactions
- Makes it possible for transactions with the same overlapping access sets to proceed in parallel
- Each transaction $t$ logs the set of transactions that have made uncommitted writes that transaction $t$ viewed
- Transaction $t$ aborts if any of its dependencies abort



# Multiversioned timestamp ordering

- Tool for tracking object versions with concurrent execution
- Idea: each transaction sees a consistent snapshot of the database that existed when the transaction started
- Each transaction associated with a timestamp
- Objects associated with a version chain
- Read: fetch the object's latest version with timestamp < transaction timestamp, record time of last access
- Write: update object version and record transaction timestamp, will fail if transaction's timestamp < timestamp of last access

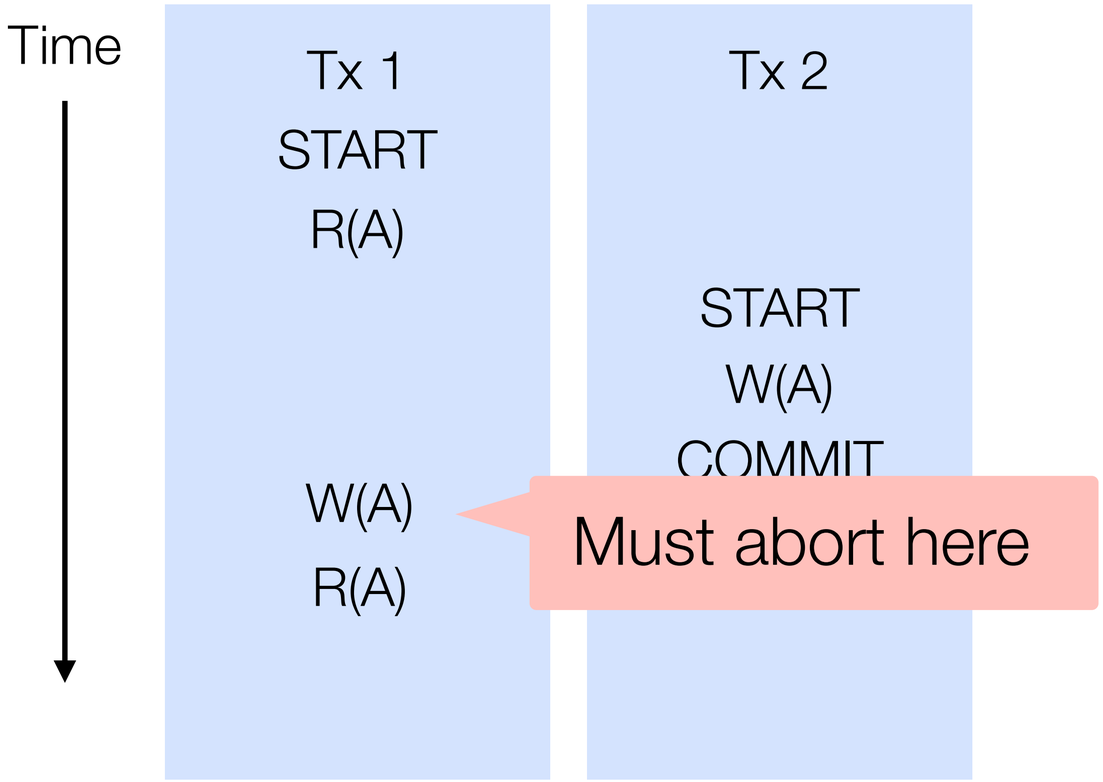
# Multiversioned timestamp ordering




Can’t write if a future transaction has accessed the object

No violations: safe to commit

# Multiversioned timestamp ordering



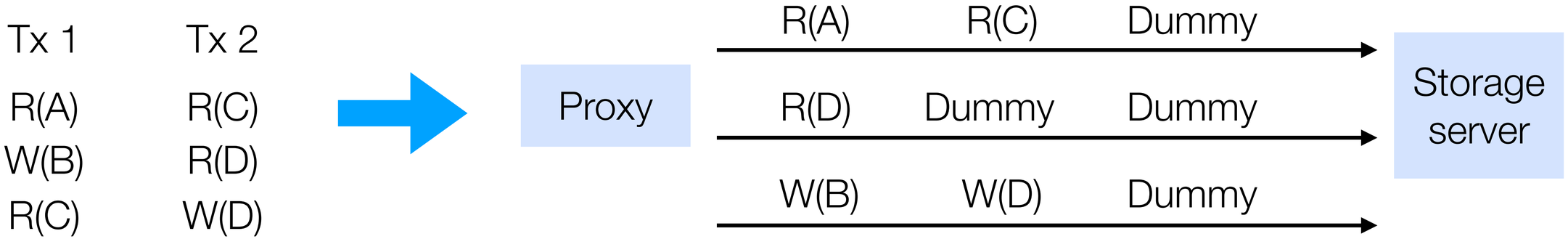

Can’t write if a future transaction has accessed the object

# Issuing read and write requests

- Given an object version, need to issue ORAM requests
- Leverage fact that transactions can be rewritten to run reads before writes
- Read phase
  - At fixed intervals, send fixed-size read batches (pad with dummy requests)
  - Deduplicates read operations that access same object
- Write phase
  - During execution, proxy buffers write operations in a version cache
  - At the end of the epoch, latest versions of each object in version cache are written in fixed-size write batch
- Deterministic structure hides transaction structure

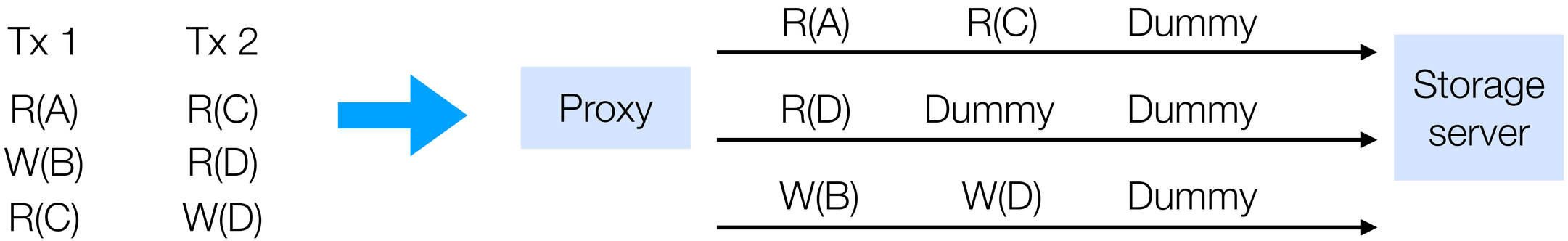
# Executing transaction reads/writes

- Read phase:
  - At fixed intervals, send fixed-size read batches (pad with dummy requests)
  - Deduplicates read operations that access same object
- Write phase
  - During execution, proxy buffers write operations in a version cache
  - At the end of the epoch, latest versions of each object in version cache are written in fixed-size write batch



# Splitting epochs into read/write phases

- Server learns number of read batches and the size of each of the batches
- These parameters need to be fixed across epochs for security
- Need to choose parameters with a particular application in mind:
  - Batches are too large: waste resources
  - Batches are too small: frequent aborts



# Optimization: reducing work for read/write accesses

- Differentiate between “logical” cache and ORAM stash
  - Logical cache contains objects that were stored as a result of a logical access
  - ORAM stash is maintained via Ring ORAM algorithm and contains both items that are logically fetched and other items on the tree path
- Safe to skip accesses to logical cache (and pad to fixed number of requests with dummies), but not to ORAM stash
  - Skipping accesses to ORAM stash would affect distribution of requests (paths would skew away from latest evicted paths)
- ORAM server can learn if a request is a read or write because made in deterministic batches: writes can happen without corresponding read operation

# Obladi’s approach

Support concurrent transactions:

- How to maintain consistency while running transactions simultaneously?
- **How to exploit parallelism for performance?**

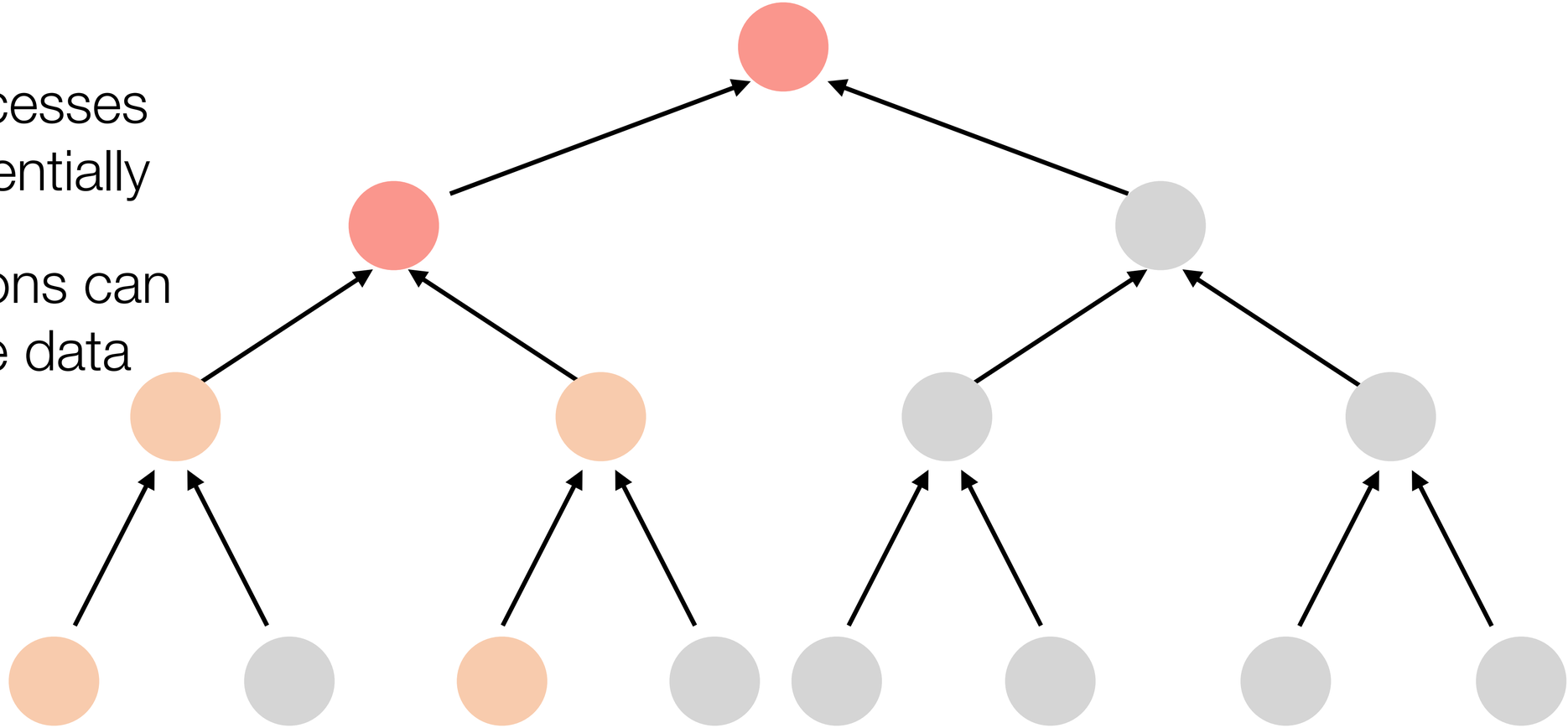
Durability: Ensure committed transactions are recorded even if there’s a crash

# Parallelizing ORAM

Want to execute operations in parallel

Challenge?

- Ring ORAM processes operations sequentially
- Different operations can access the same data



# Obladi parallelism

Approach: run accesses in parallel where there is no contention

Observations

- Operations access mostly disjoint buckets
- When accessing the same bucket, accesses to the same bucket are for different blocks (before eviction or reshuffling operations)

Can avoid requests to the same object by deduplicating requests in a batch

# Obladi parallelism

Challenge: Ring ORAM evictions assume requests occur sequentially

Idea: break up physical reads and physical writes

- Read: compute metadata and execute set of physical reads for logical read paths, early reshuffle, and evict operations → appears random

- Writes: flushed at the end of the epoch (buffer intermediate writes locally) → locations always deterministic

# Obladi’s approach

Support concurrent transactions:

- How to maintain consistency while running transactions simultaneously?
- How to exploit parallelism for performance?

**Durability: Ensure committed transactions are recorded even if there’s a crash**

# Obladi durability

If a transaction commits, then the effects of the transaction must be preserved, even if there are failures at the clients, proxy, or cloud storage

Challenge: durability while hiding access patterns

Approach: Fate sharing

- Either all transactions in an epoch are made durable, or none are
- In the event of a failure, revert state to the previous epoch

# Recovering from a failure

If a crash occurs, Obladi must:

- Recover proxy metadata lost during the crash
- Make sure that the ORAM doesn't include aborted transactions' updates

Before committing, proxy logs the position map, permutation map, and stash

- Encrypted, padded to the max size, and stored at the server

Server needs to keep copies of old data so that it can roll back

- Server uses copy-on-write: instead of updating buckets in place, creates new versions of each bucket
- Evictions are deterministic, and so on a failure, can revert by rolling back evict path counter and reverting to corresponding old object versions

# Client retries after failure

Challenge: After a failure, client may retry an operation

- If a transaction accessing object x aborted due to a proxy failure, the client will likely attempt to access x again
- New operation for object x will map to the same path — reveals that access is to a real object rather than a dummy!

Solution:

- Before making requests, durably log all accessed locations
- At recovery time, replay the requests for aborted transactions

# Outline

1. Ring ORAM
2. Transactions
3. Obladi
4. **Horizontally scalable ORAM**
5. Logistics

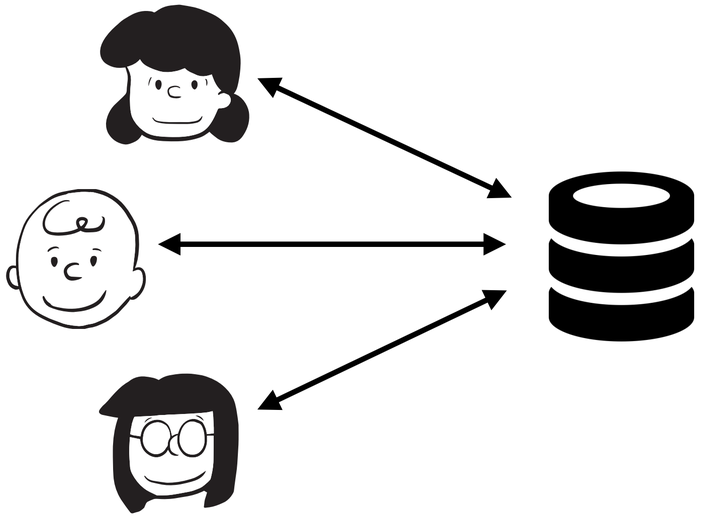
# Discuss: Limitations on scaling ORAM

What are barriers to scaling existing ORAM systems to support more requests?

# Existing ORAM schemes have scalability bottlenecks

Scalability bottleneck:

- Coordination required for every request
- Cannot securely distribute



# Existing ORAM schemes have scalability bottlenecks

Scalability bottleneck:

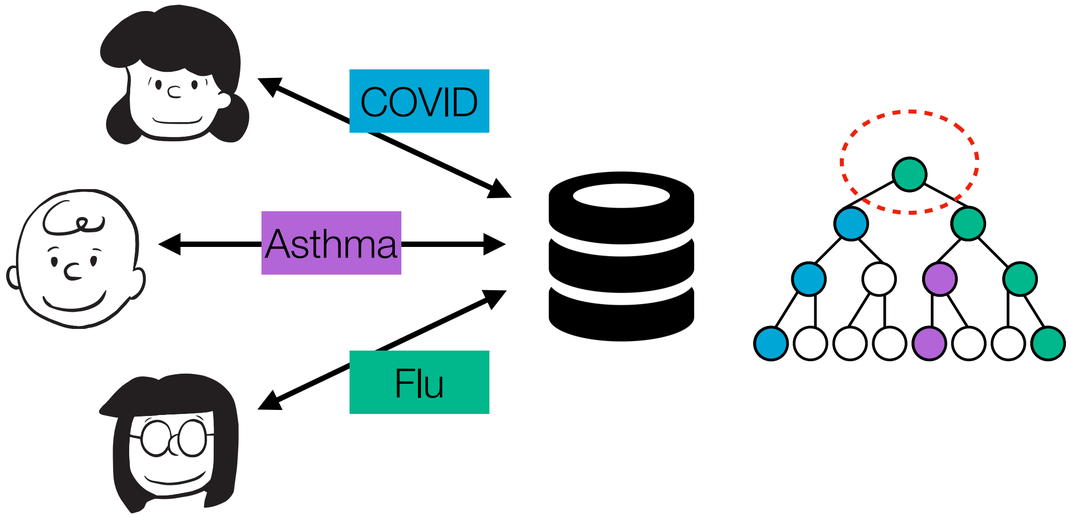
- Coordination required for every request
- Cannot securely distribute

Most systems are tree-based and
hide locations of objects in the tree.

Common bottlenecks:

-Location metadata

-Tree root



# Existing ORAM schemes have scalability bottlenecks

Scalability bottleneck:

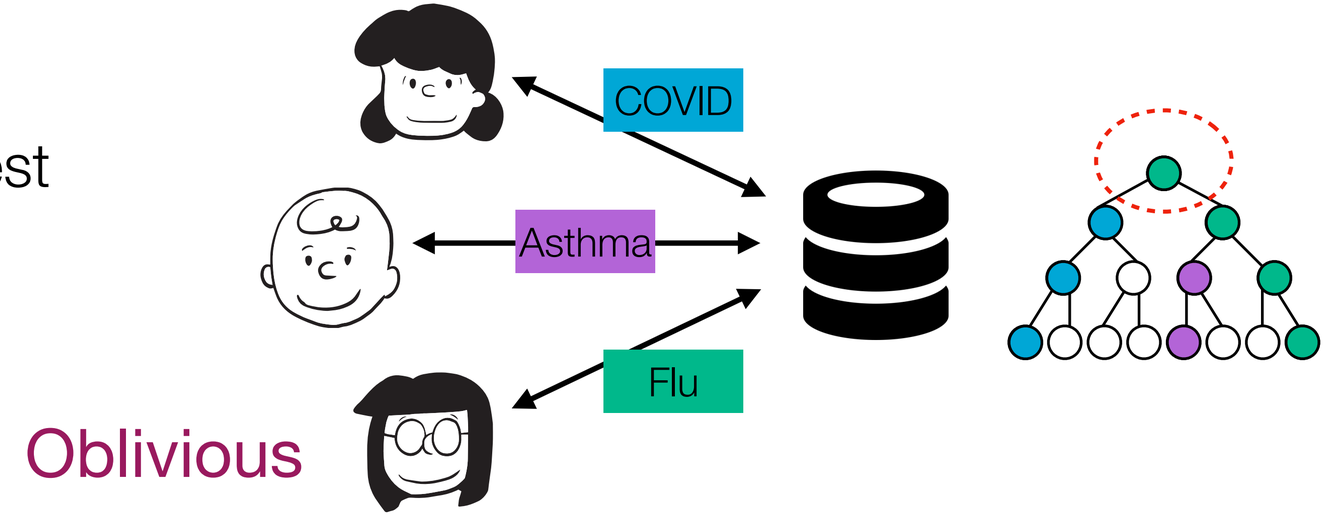
- Coordination required for every request
- Cannot securely distribute

Most systems are tree-based and hide locations of objects in the tree.

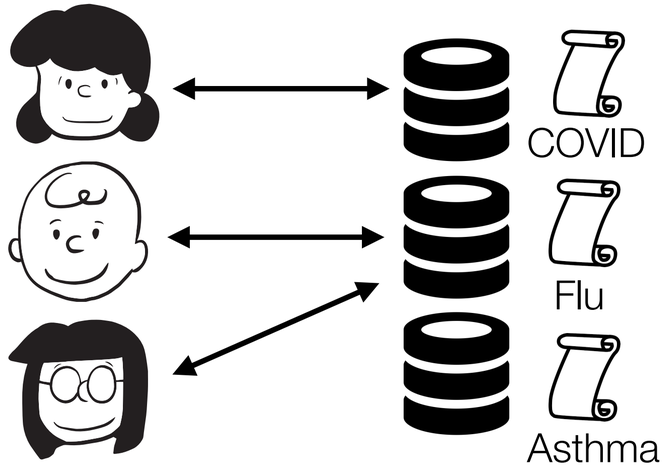
Common bottlenecks:

-Location metadata

-Tree root




Oblivious

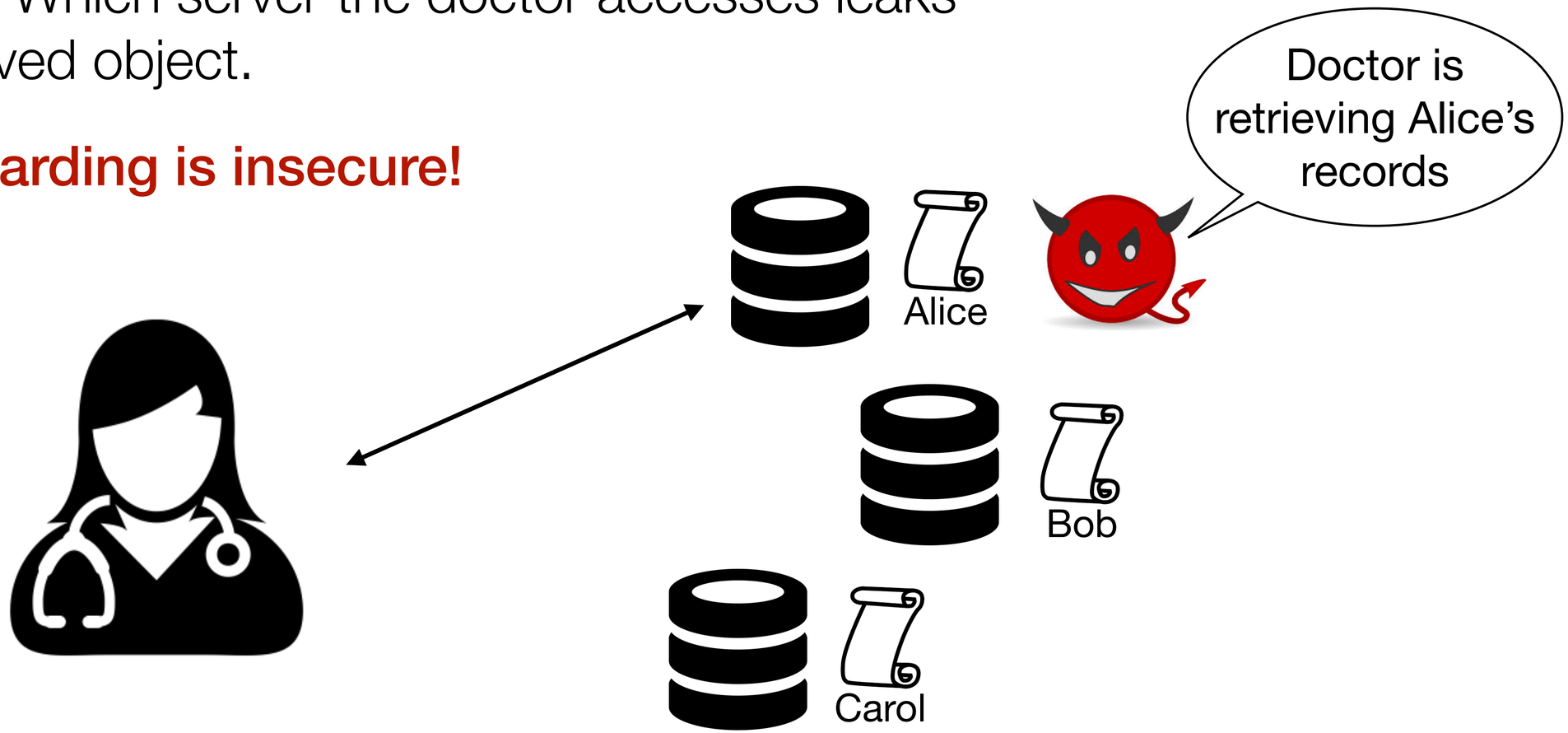



Plaintext

# Attempt #1: Shard objects across servers

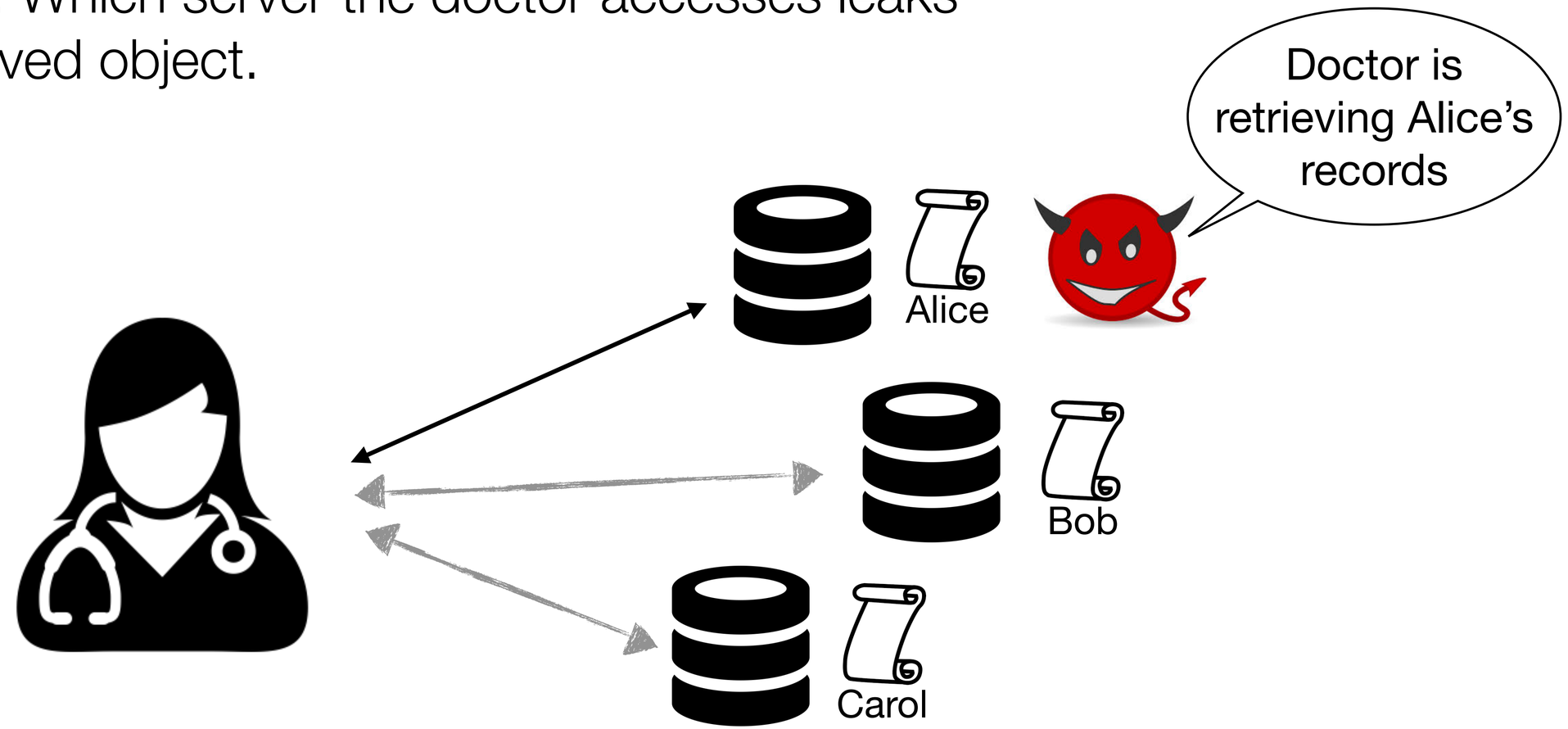
Problem: Which server the doctor accesses leaks the retrieved object.

**Naive sharding is insecure!**



# Attempt #2: Dummy request to every server

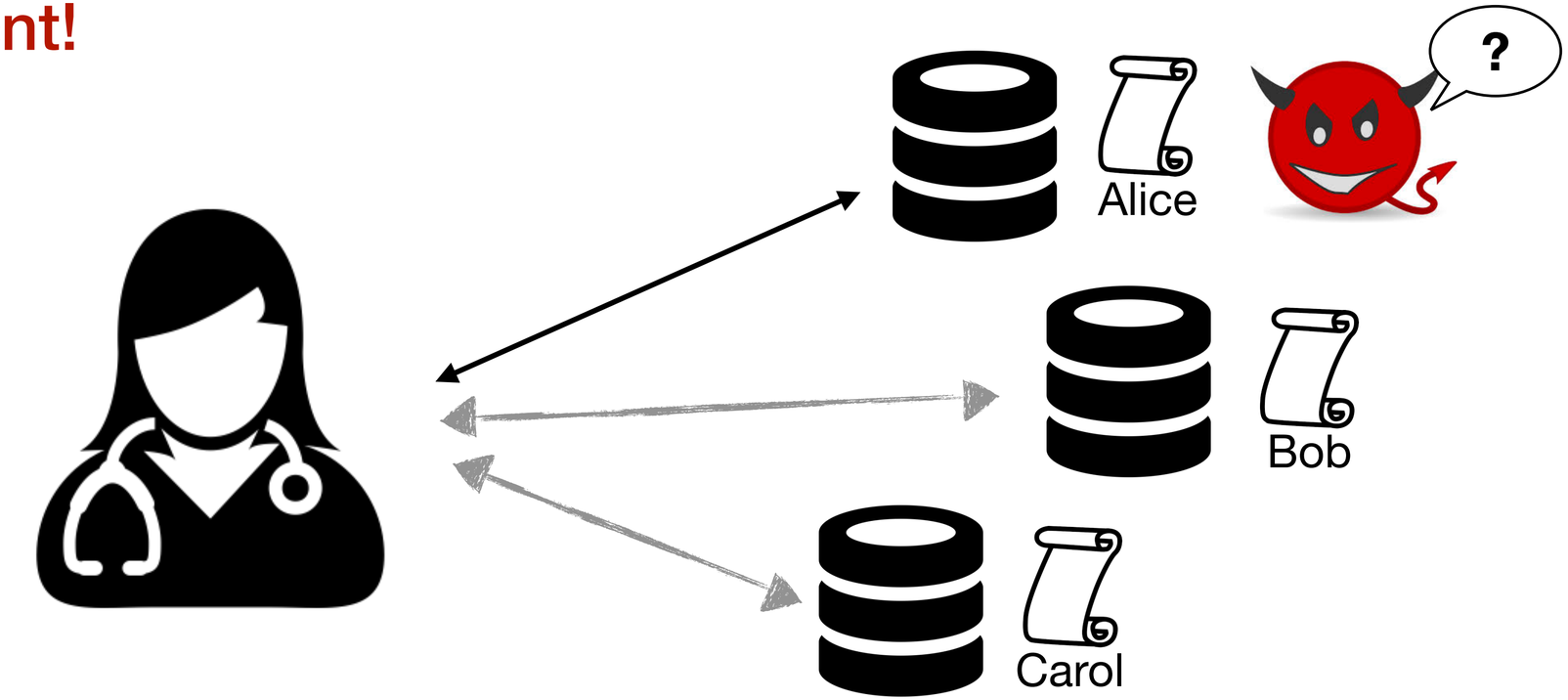
Problem: Which server the doctor accesses leaks the retrieved object.



# Attempt #2: Dummy request to every server

Problem: ~~Which server the doctor accesses leaks the retrieved object.~~

**Inefficient!**



# Snoopy: oblivious object store that scales like plaintext storage

[Dauterman, Fang, Demertzis, Crooks, Popa]

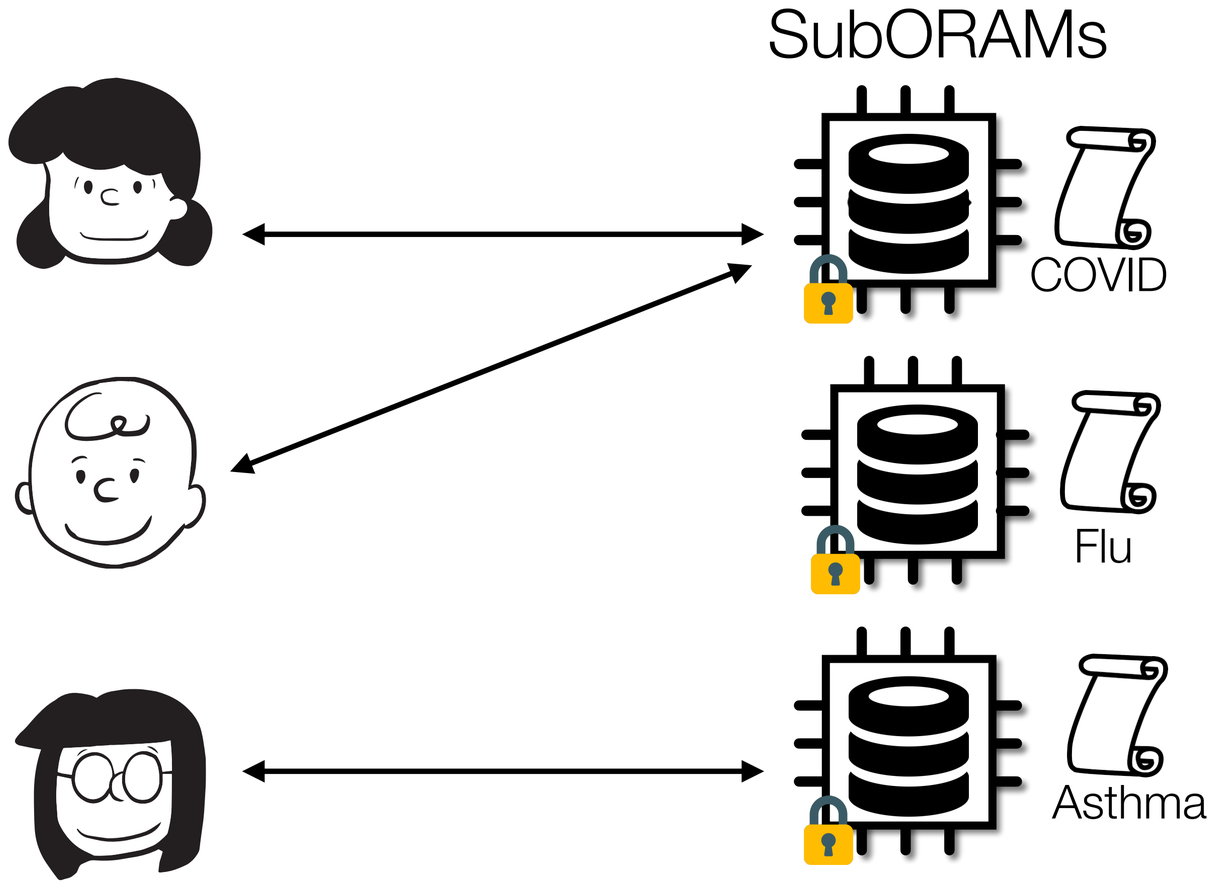
- Add more machines → throughput increases
- Uses hardware enclaves to support multiple users without relying on a trusted proxy
  - Not only do storage accesses need to be oblivious, but also accesses to client state

# Building Snoopy

## Classic techniques

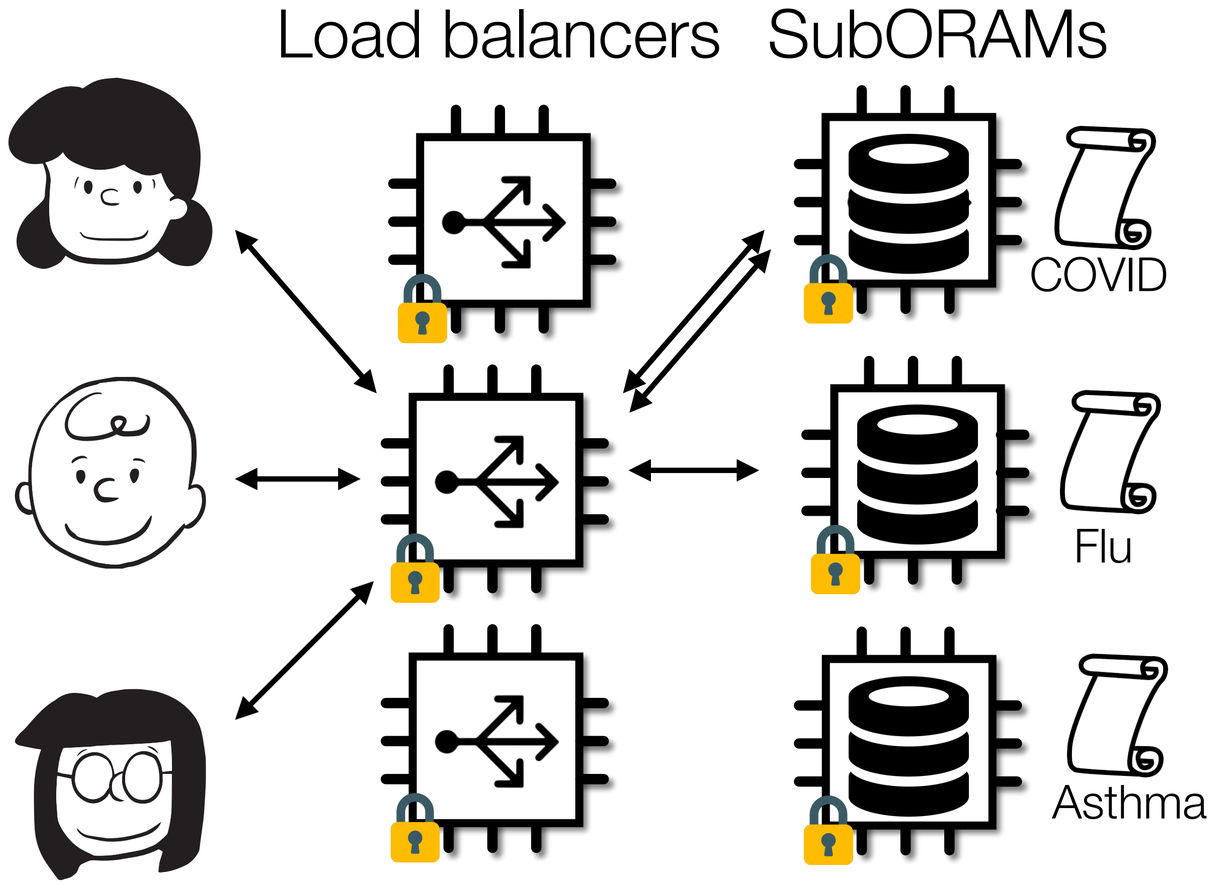
# Building Snoopy

**Classic techniques**
Partitioning



# Building Snoopy

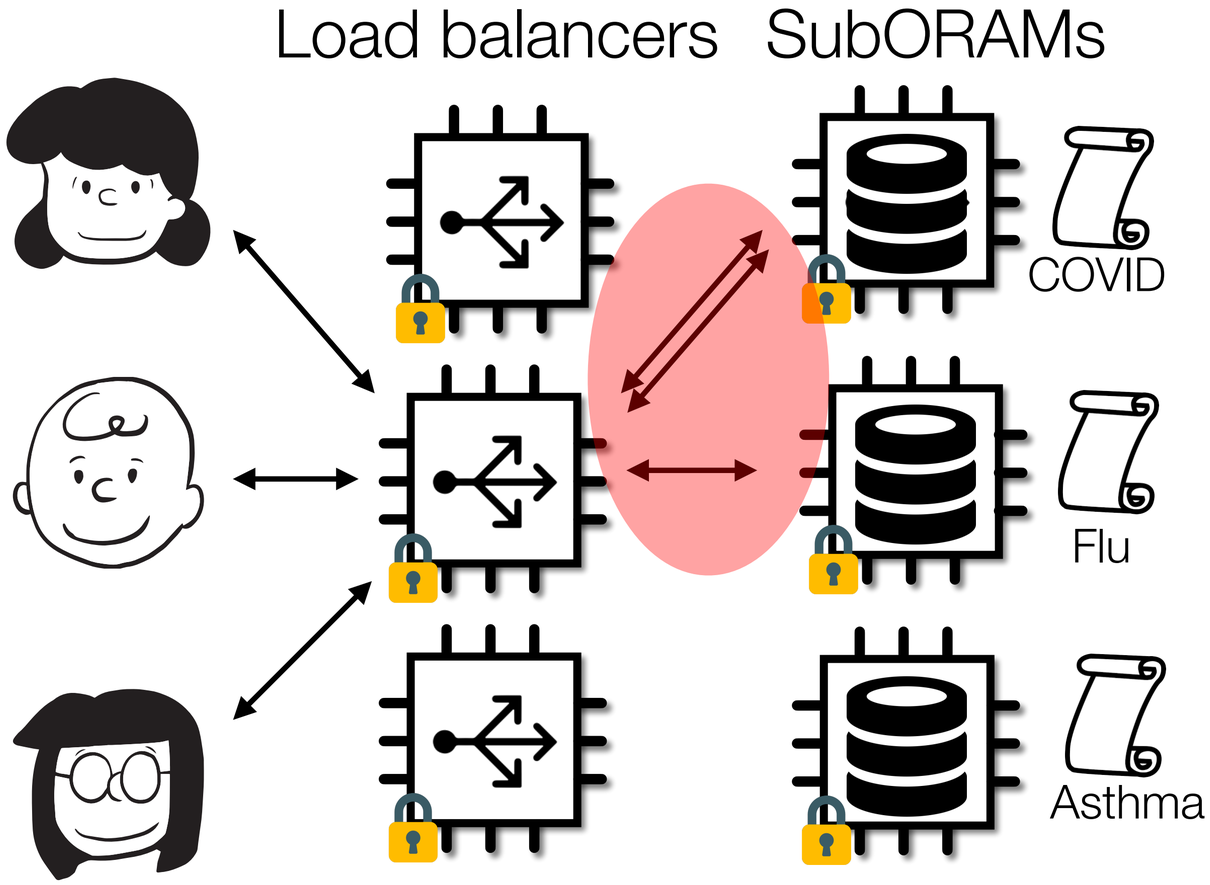
**Classic techniques**
Partitioning
Batching



# Building Snoopy

**Classic techniques**
Partitioning
Batching

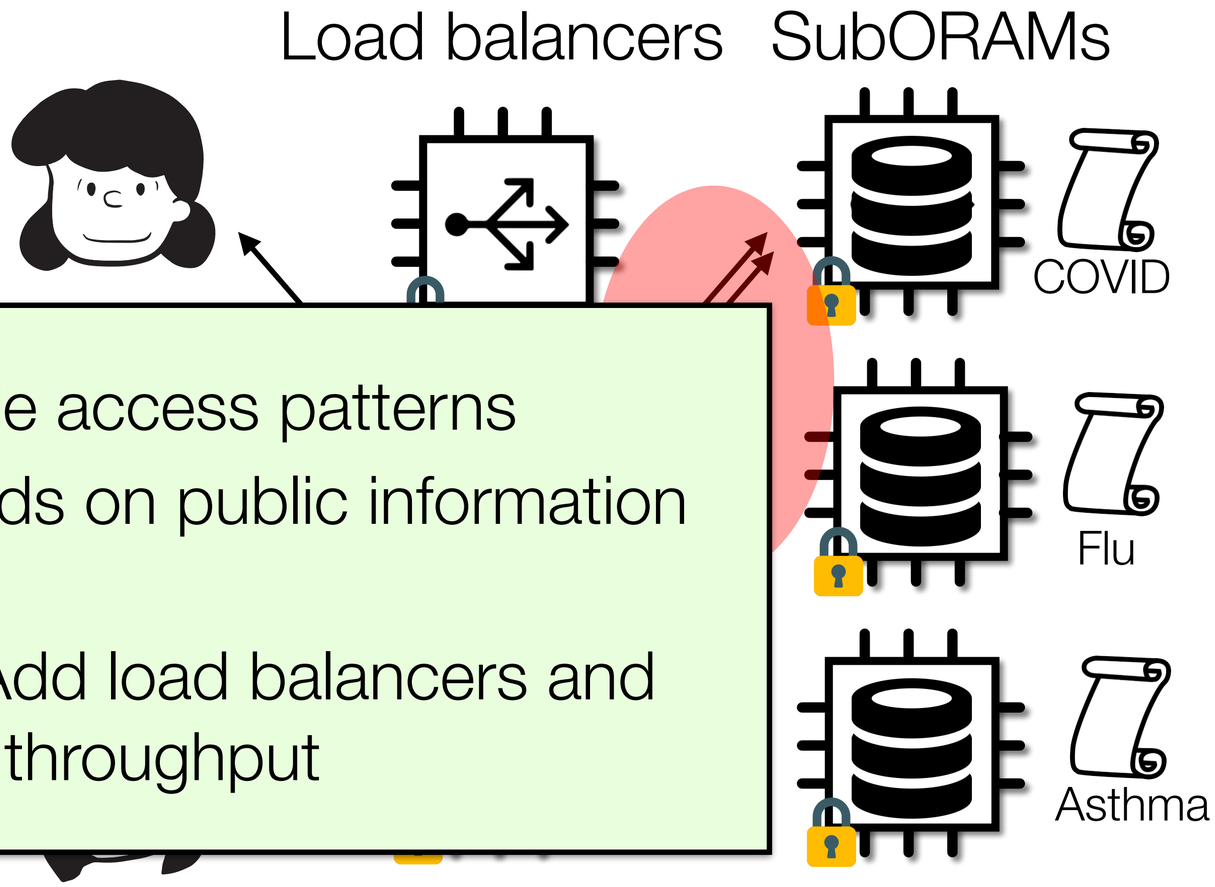
😨 **Naively insecure**
Batches sent to subORAMs reveals request distribution



# Building Snoopy

**Classic techniques**
Partitioning
Batching

😨 **Naively insecu**
Batches sent to su
request distributior




**Goal #1 (Security):** Hide access patterns
-Batch size only depends on public information

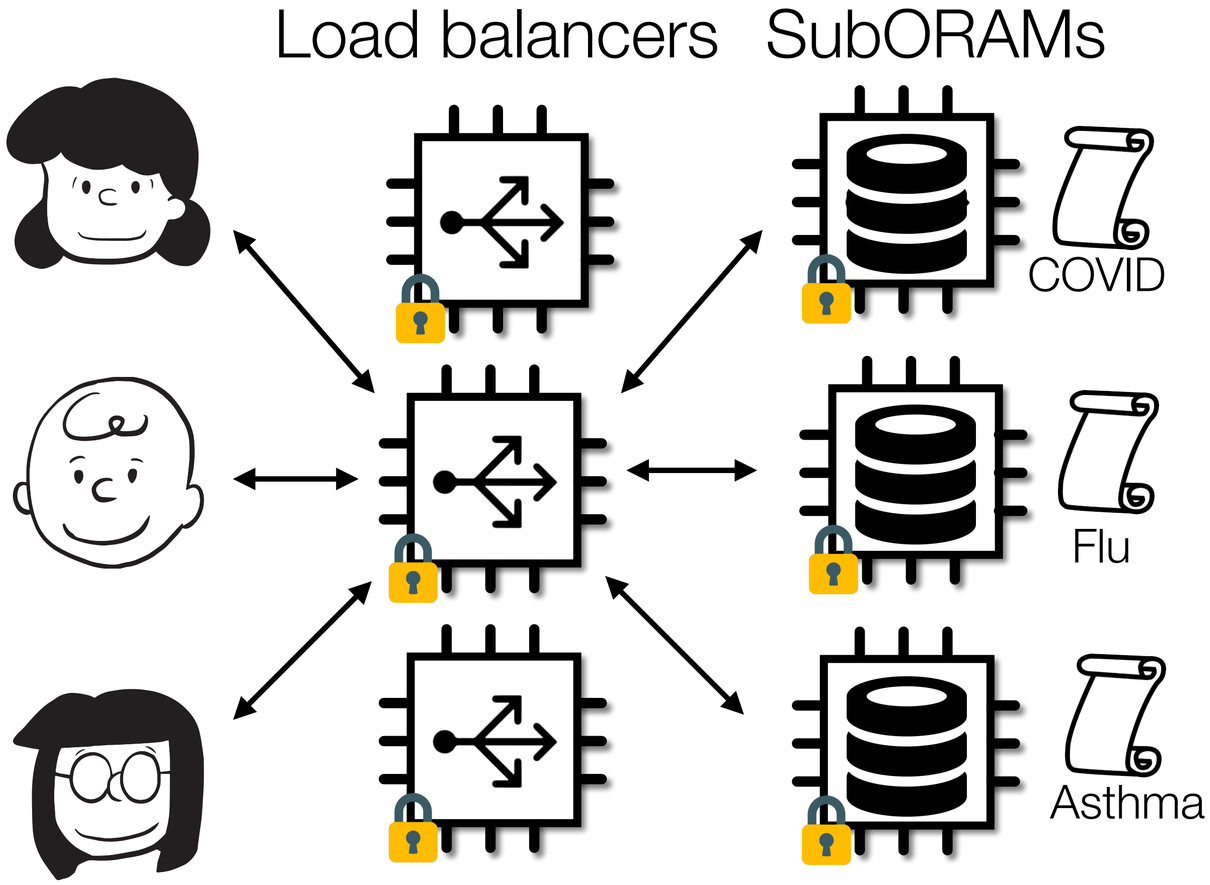
**Goal #2 (Scalability):** Add load balancers and subORAMs to increase throughput

# Building Snoopy

**Classic techniques**
Partitioning
Batching

😨 **Naively insecure**
Batches sent to subORAMs reveals request distribution

💡**Contributions**
Techniques that enable *batching* + *partitioning* with *security* + *scalability*



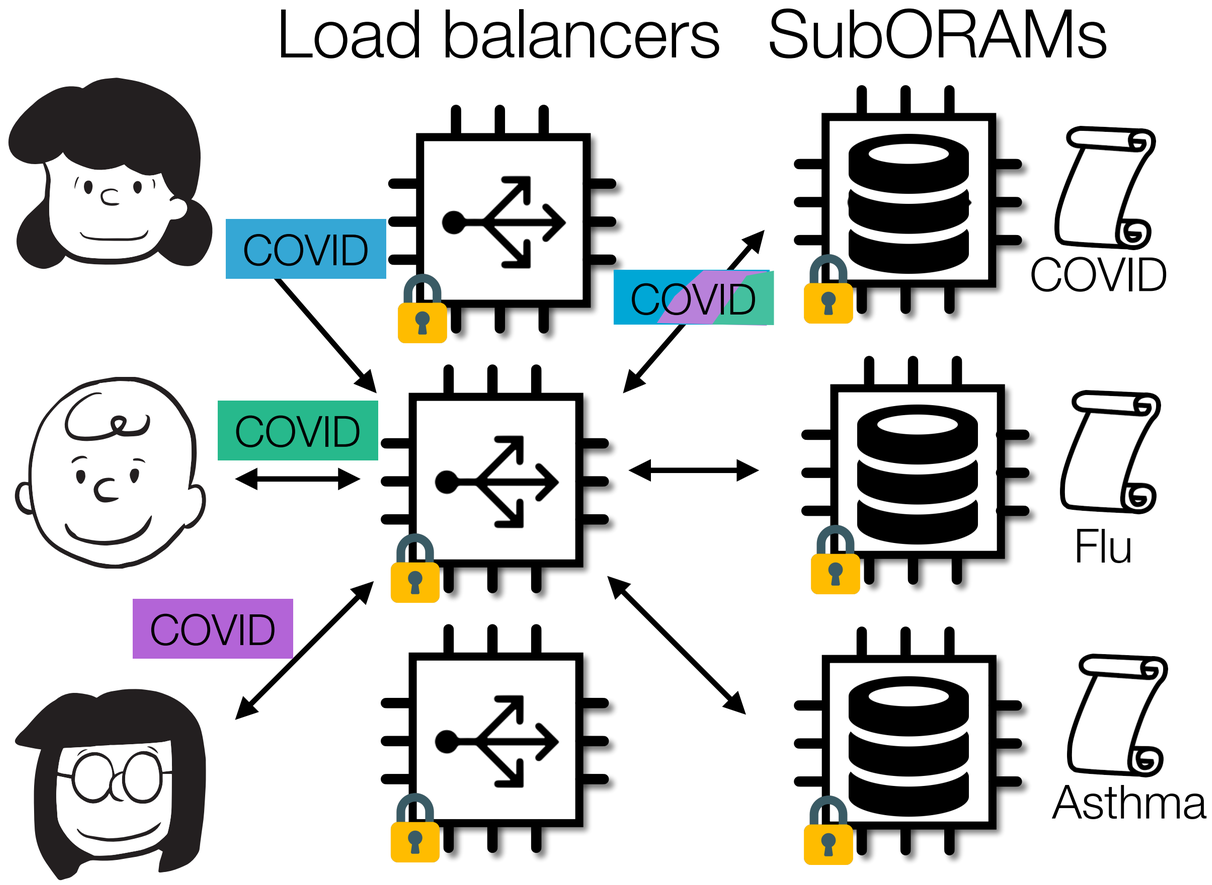
# Handling skewed workloads

If every client requests the same object, then batch size = total requests

→ not scalable!

💡 Deduplication

Now we only need to handle **distinct** requests.



# Securely setting batch size $B$

**Requirements**

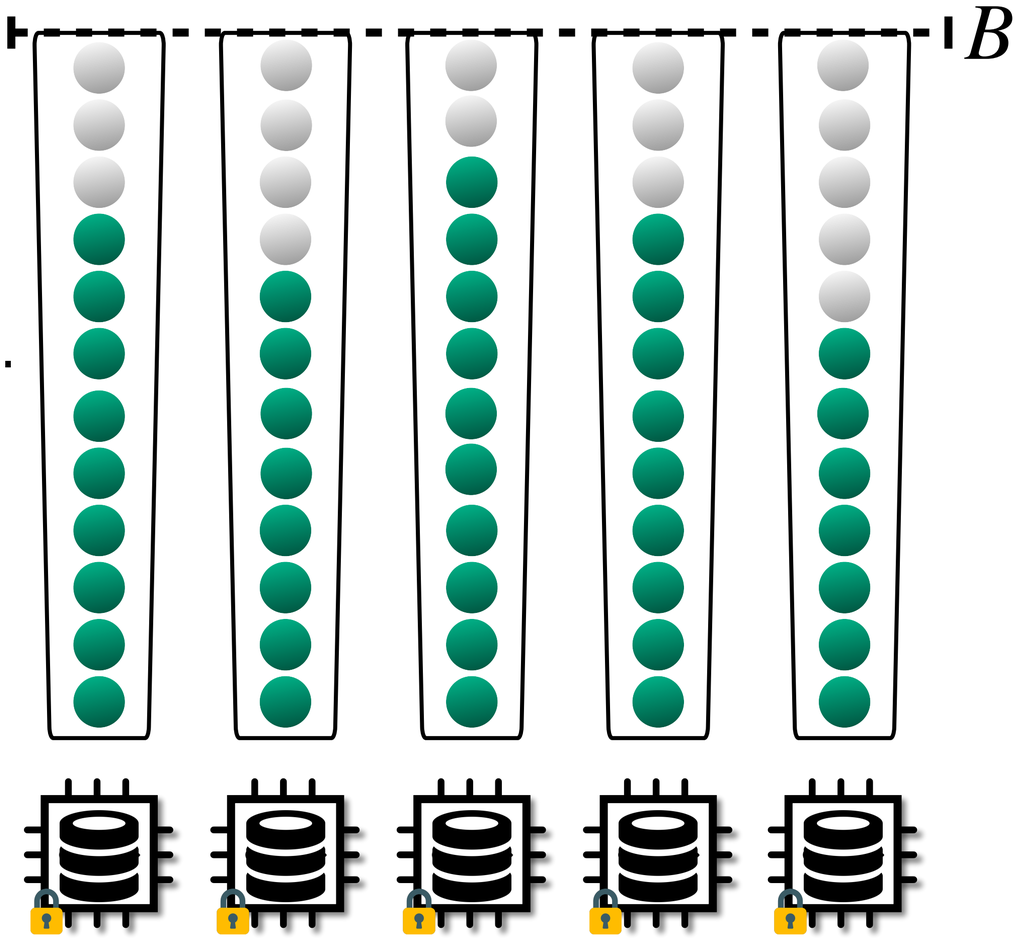
-Computable with public information

-Negligible overflow probability

**High-throughput** → many concurrent requests.

After deduplication, requests are spread across subORAMs (≈) evenly.

💡Don’t need to add many dummy requests to have secure batch size.



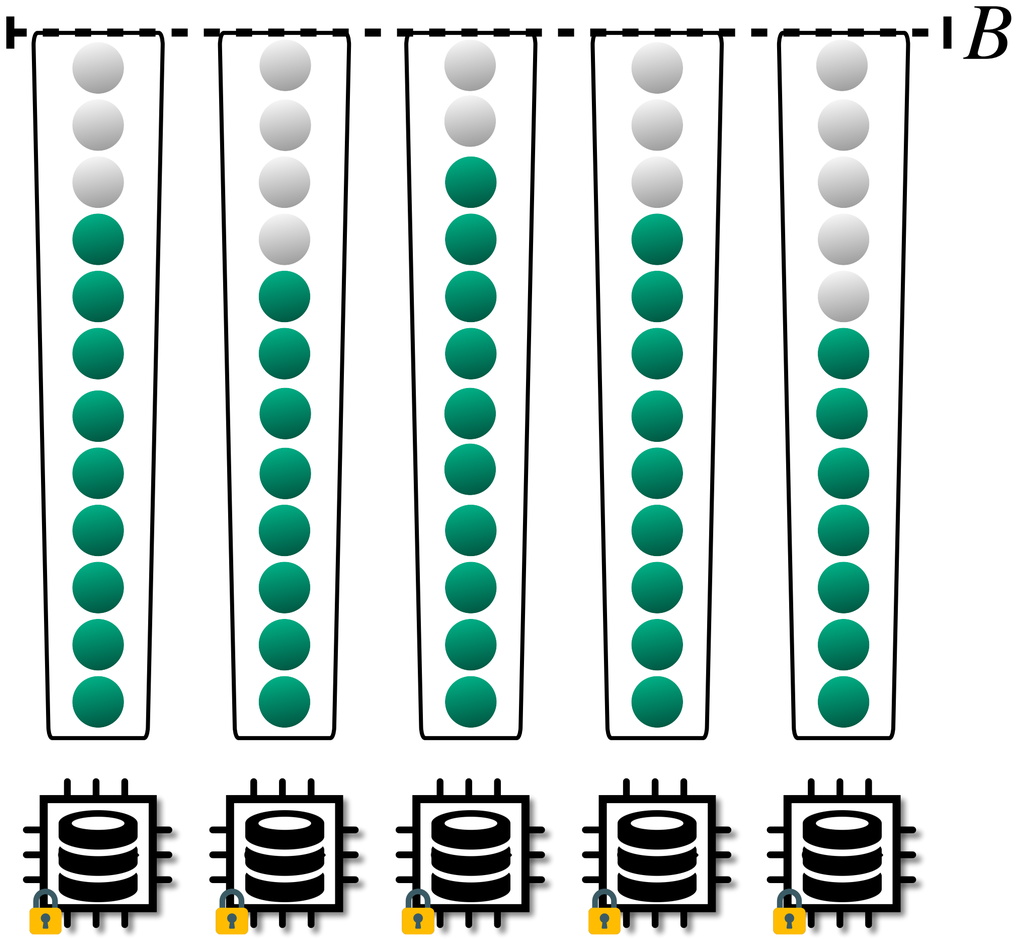
# Securely setting batch size $B$

**Requirements**

-Computable with public information

-Negligible overflow probability

Can model as a balls-into-bins problem.



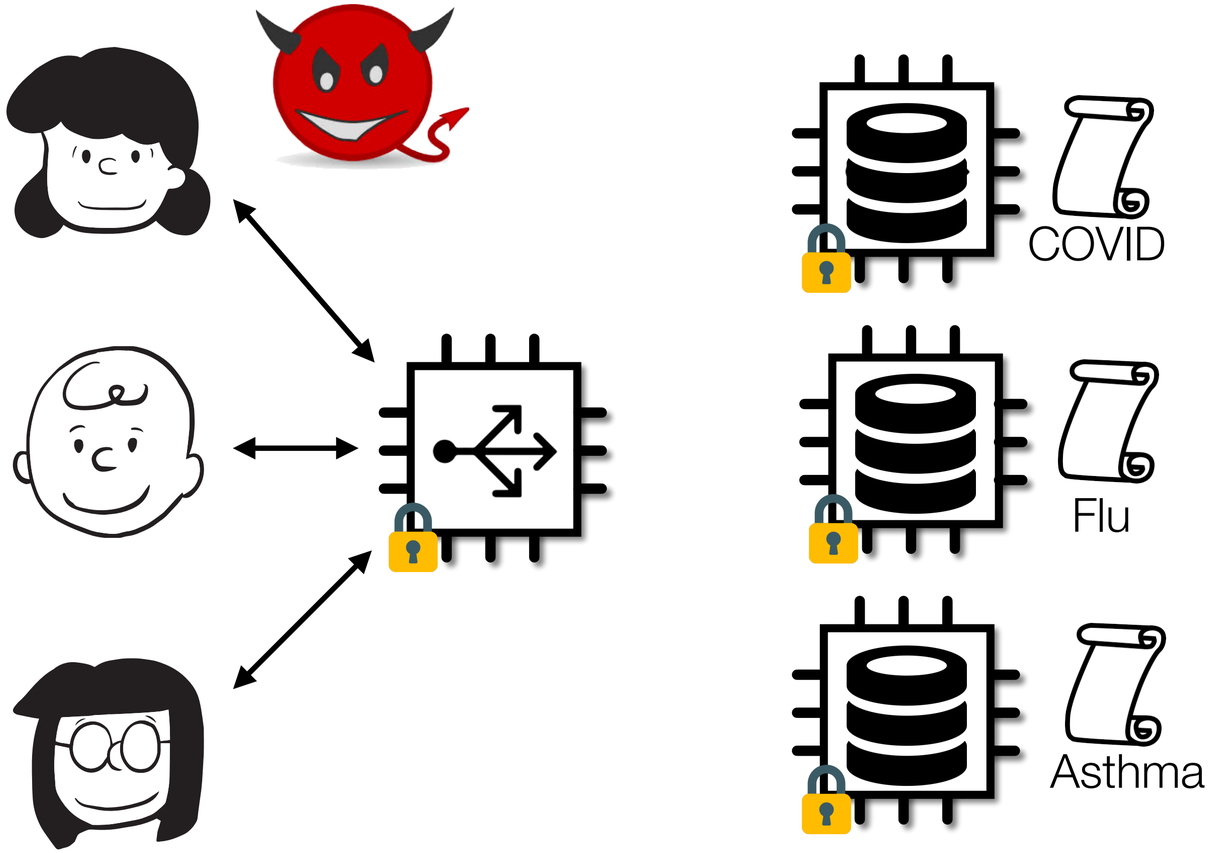
# Attacker cannot cause overflow (with high probability)

Attacker's goal: Overflow request batch

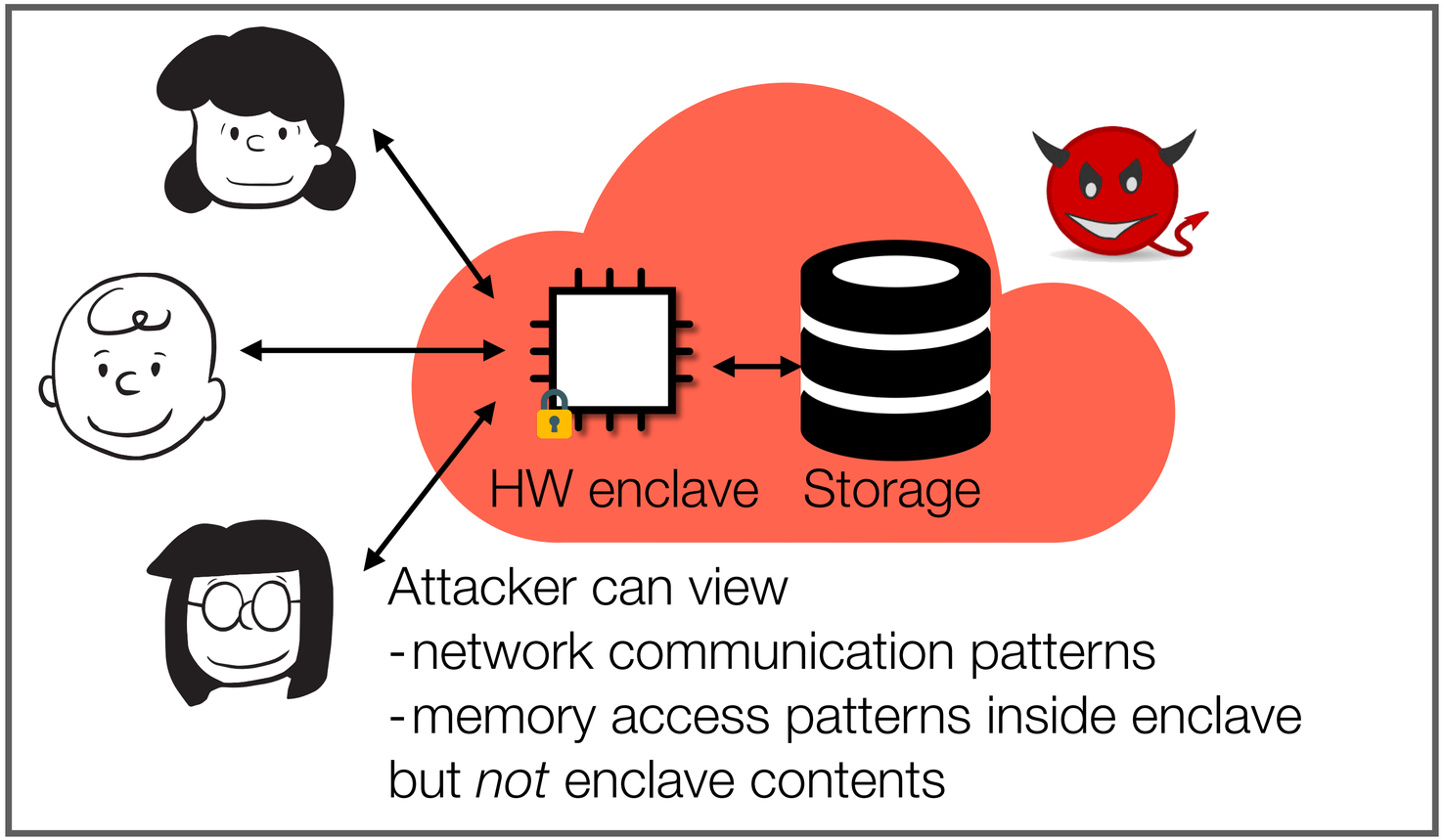
Snoopy's defenses:

- Deduplication (identical requests ⇏ overflow)
- Hidden mapping of requests to subORAMs (keyed hash)
- Oblivious request routing

By balls-into-bins analysis, attacker cannot overflow with high probability.



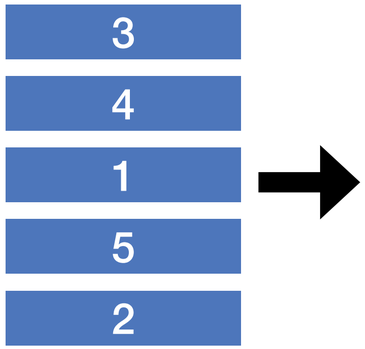
# Designing oblivious algorithms




Memory access patterns should not leak information about requests.

# Oblivious building blocks

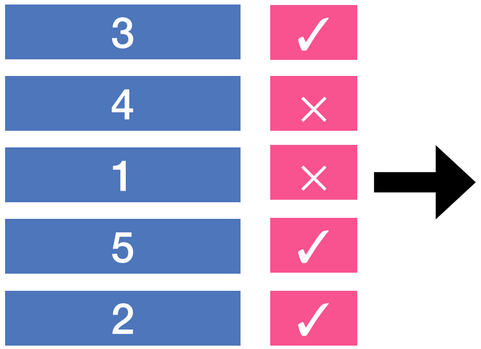
Perform compare-and-swaps in fixed, predefined order




Oblivious sort

$O(n \log^2 n)$

[Batcher68]

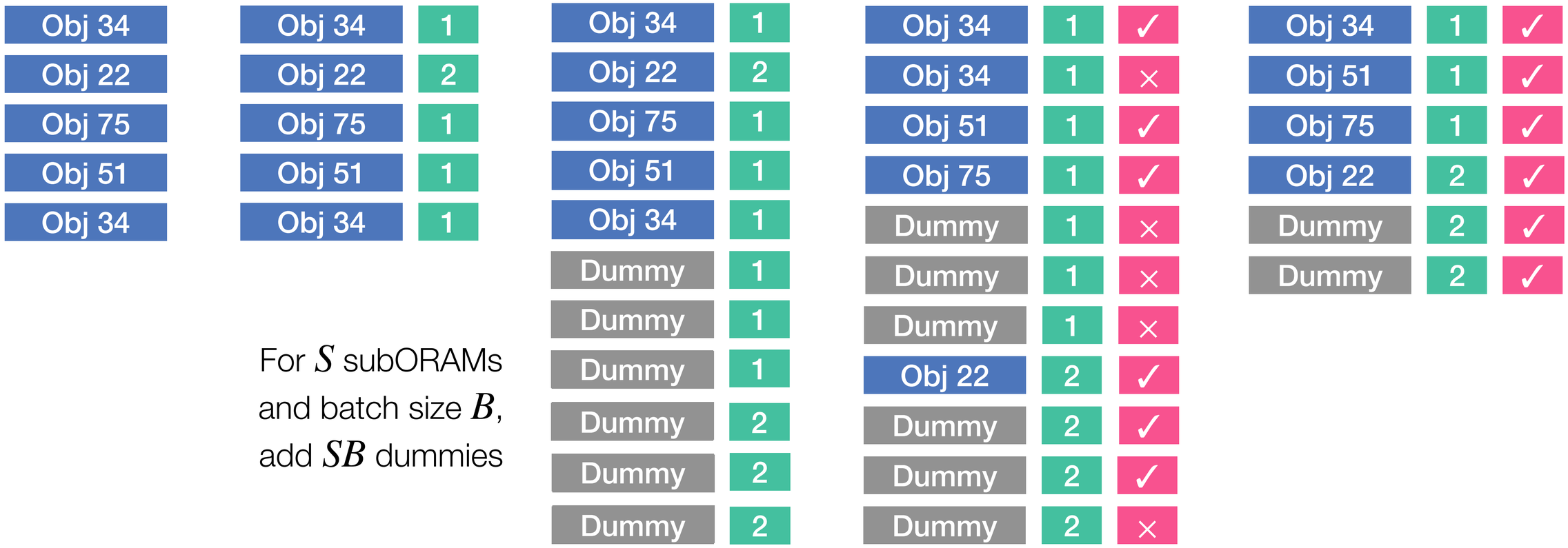



Oblivious compaction

$O(n \log n)$

[Goodrich11]

# Constructing batches obliviously




1. Assign requests to subORAMs
2. Add dummy requests
3. OSort to construct batches with extra dummies
4. OCompact out extra dummies.

# Matching subORAM responses to client requests

Same key ideas from constructing batches (see paper for details)

Need to:

- Filter out dummies
- Propagate subORAM responses to potentially multiple client requests

# Outline

1. Ring ORAM
2. Transactions
3. Obladi
4. Horizontally scalable ORAM
5. **Logistics**

# Logistics

Final project proposal due next class! (10/16)

Come to office hours, talk to me after class, or email me if you want to talk about ideas.

Look out for sign ups for meetings to discuss project proposal

Note: schedule change starting 11/4 that will affect some student presentation dates

# References


Crooks, Natacha, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. "Obladi: Oblivious serializable transactions in the cloud." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 727-743. 2018.

Dauterman, Emma, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. "Snoopy: Surpassing the scalability bottleneck of oblivious storage." In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 655-671. 2021.

Ren, Ling, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. "Constants count: Practical improvements to oblivious {RAM}." In *24th USENIX Security Symposium (USENIX Security 15)*, pp. 415-430. 2015.

Stefanov, Emil, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: an extremely simple oblivious RAM protocol." *Journal of the ACM (JACM)* 65, no. 4 (2018): 1-26.

https://15445.courses.cs.cmu.edu/fall2023/slides/17-timestampordering.pdf