A Survey of Oblivious RAMs

David Cash

IBM

Securely Outsourcing Memory

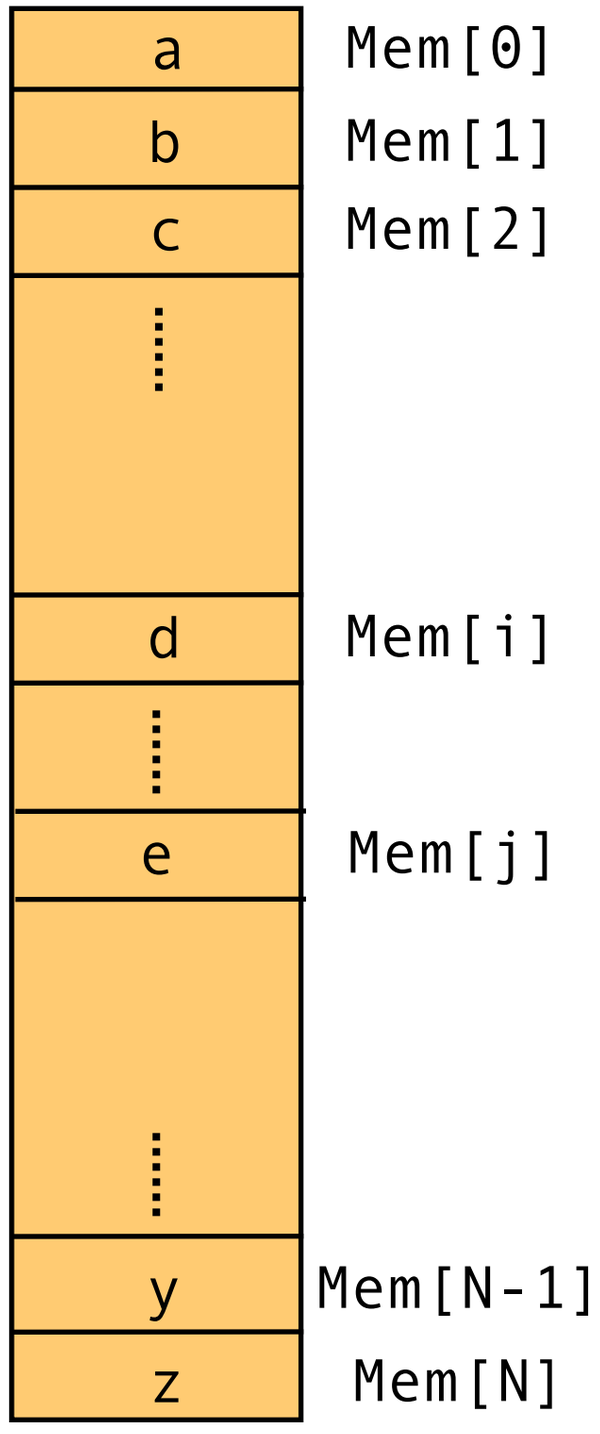
Goal: Store, access, and update data on an **untrusted** server.

Client

Write(i,x)

Read(j)

Server

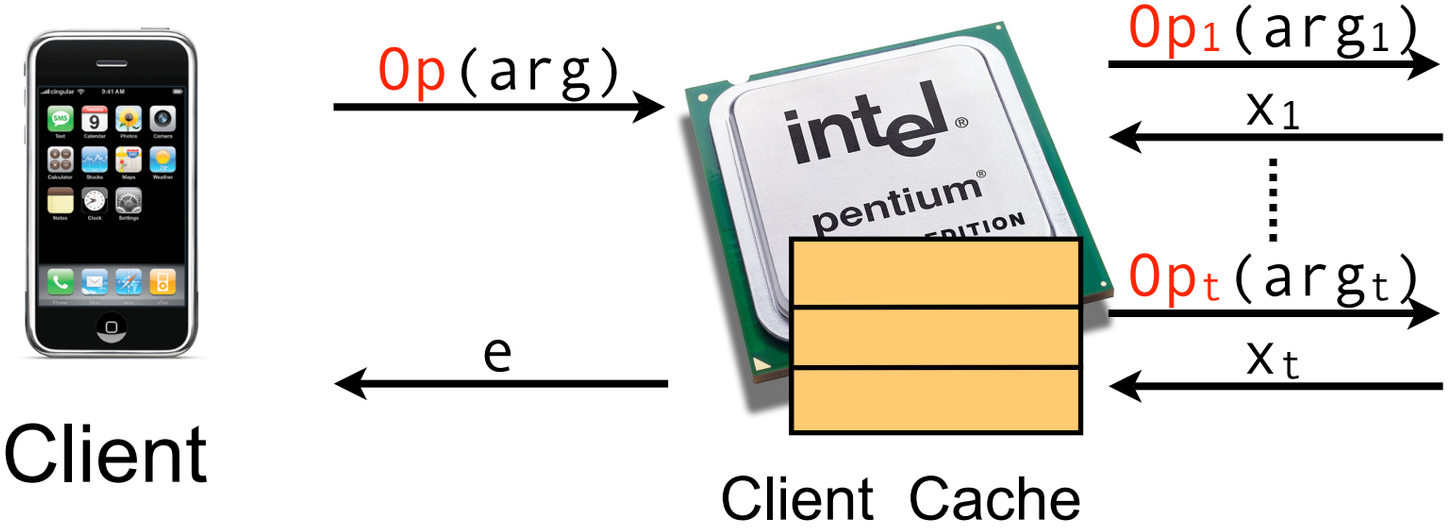


"Untrusted" means:

- It may not implement Write/Read properly
- It will try to learn about data

Oblivious RAMs

An **ORAM emulator** is an intermediate layer that protects any client (i.e. program).



ORAM will issue operations that deviate from actual client requests.

Correctness: If server is honest then input/output behavior is same for client.

Security: Server cannot distinguish between two clients with same running time.

Simplifying Assumptions

Assumption #1: Server does not see data.

↳ Store an encryption key on emulator and re-encrypt on every read/write.

Assumption #2: Server does not see op (read vs write).

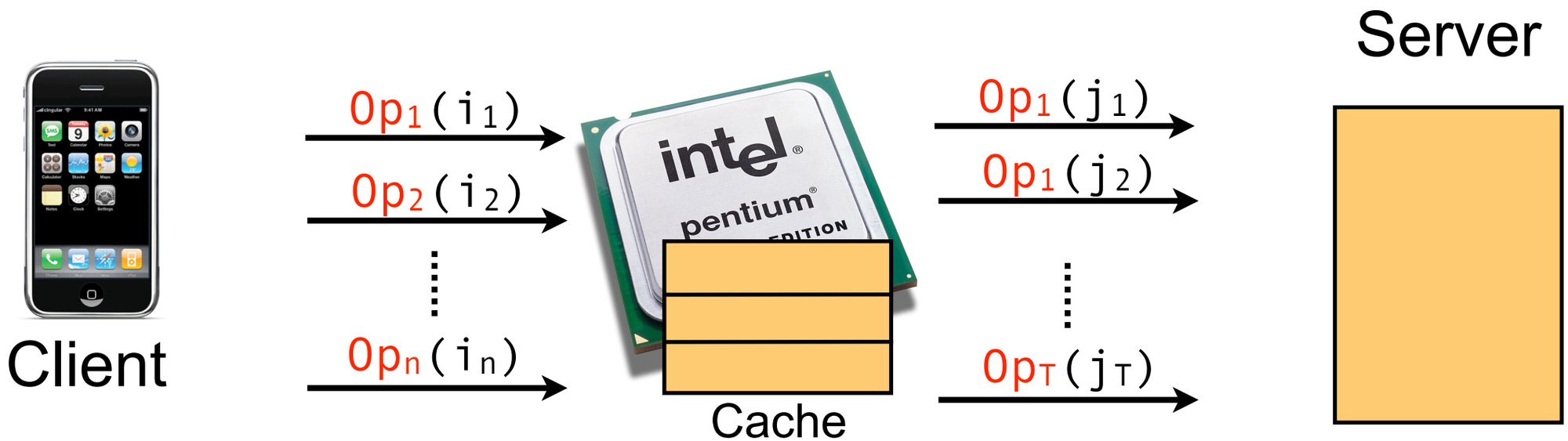
↳ Every op is replaced with both a read and a write.

Assumption #3: Server is honest-but-curious.

↳ Store a MAC key on emulator and sign (address, time, data) on each op... (more on this later)

ORAM Security

What's left to protect is the "access pattern" of the program.



Definition: The **access pattern** generated by a sequence $(i_1, ..., i_n)$ with the ORAM emulator is the random variable $(j_1, ... , j_T)$ sampled while running with an honest server.

Definition: An **ORAM emulator is secure** if for every pair of sequences of the same length, their access patterns are indistinguishable.

Enforcing Honest-but-Curious Servers

Assumption #3: Server is honest-but-curious.

↳ Store a MAC key on client and sign (addr, time, data) on each op...

Simple authentication does not work: What do we check with timestamp?

It does work if scheme supports "time labeled simulation"

↳ Means system can calculate "last touched" time for each index at all times.

Then can check if server returned correct (addr, time, data)

Some of the recent papers might not support this.

Information-Theoretic ORAM

- There exist **non-trivial information-theoretically secure ORAMs**
  - *Ajtai'10* and *Damgaard, Meldgaard, Nielsen'10* gave schemes
- Mostly of interest for complexity theory, i.e. actually simulating a RAM
- For outsourcing memory, we still need cryptographic assumptions for the encryption and authentication
- Thus we ignore these less efficient schemes today

ORAM vs Private Info Retrieval (PIR)

PIR: Oblivious transfer without sender security (i.e. receiver may learn more than requested index)

Some differences:

In ORAM...	In PIR...
Server data changes with each operation	Server data does not change
Server only performs simple read/write ops	Server performs “heavier” computation
Client may keep state between queries	Client does not keep state

ORAM Efficiency Measures

Parameter: N - number of memory slots

Efficiency measures:

- **Amortized overhead**: # ops issued by ORAM simulator divided by # of ops issued by client
- **Worst-case overhead**: max # ops issued by ORAM simulator to respond to any given call by program
- **Storage**: # of memory slots used in server
- **Client storage**: # of slots stored in ORAM emulator between ops
- **Client memory:** max # of slots used in temp memory during processing of an op

Can also look at scaling with size of memory slots. (Not today)

Uninteresting ORAMs

Example #1: Store everything in ORAM simulator cache and simulate with no calls to server.

↳ **Client storage = N.**

Example #2: Store memory on server, but scan entire memory on every operation.

↳ **Amortized and worst-case communication overhead = N.**

Example #3: Assume client accesses each memory slot at most once, and then permute addresses using a PRP.

↳ **Essentially optimal, but assumption does not hold in practice.**

Lower Bounds

Theorem (GO'90): Any ORAM emulator must perform $\Omega(t \log t)$ operations to simulate t operations.

↳ Proved via a combinatorial argument.

Theorem (BM'10): Any ORAM emulator must either perform $\Omega(t \log t \log \log t)$ operations to simulate t operations or use storage $\Omega(N^{2-o(1)})$ (on the server).

↳ They actually prove more for other computation models.

ORAM Efficiency Goals

In order to be **interesting**, an ORAM must simultaneously provide

- o(N) client storage
- o(N) amortized overhead
- Handling of repeated access to addresses.

Desirable features for an "optimal ORAM":

- O(log N) worst-case overhead
- O(1) client storage between operations
- O(1) client memory usage during operations
- "Stateless client": Allows several clients who share a short key to obliviously access data w/o communicating amongst themselves between queries. Requires op counters.

History of ORAMs

- Pippenger and Fischer showed “oblivious Turing machines” could simulate general Turing machines
- Goldreich introduced analogous notion of ORAMs in ’87 and gave first interesting construction
- Ostrovsky gave a more efficient construction in ’90
- ... 20 years pass, time sharing systems become “clouds” ...
- Then a flurry of papers improving efficiency: ~10 since 2010

ORAM Literature Overview

	Nickname	Client Memory	Client Storage	Server Storage	Worst-Case Overhead	Amortized Overhead
G'87	"$\sqrt{n}$"	$O(1)$	$O(1)$	$O(n)$	$O(n \log^2 n)$	$O(\sqrt{n} \log^2 n)$
O'90	"Hierarchical"	$O(1)$	$O(1)$	$O(n \log n)$	$O(n \log^2 n)$	$O(\log^3 n)$
OS'97	"Unamortized $\sqrt{n}$"	$O(1)$	$O(1)$	$O(n)$	$O(\sqrt{n} \log^2 n)$	$O(\sqrt{n} \log^2 n)$
OS'97	"Unamortized Hierarchical"	$O(1)$	$O(1)$	$O(n \log n)$	$O(\log^3 n)$	$O(\log^3 n)$
WS'08	"Merge sort "	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n)$
GM'11	"Cuckoo 1"	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log^2 n)$
KLO'11	"Cuckoo virtual stash"	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log^2 n / \log \log n)$
GM'11	"Cuckoo 2"	$O(n^\delta)$	$O(n^\delta)$	$O(n)$	$O(n)$	$O(\log n)$
GMOT'11	"Republishing OS'97 Pt 1"	$O(1)$	$O(1)$	$O(n)$	$O(\sqrt{n} \log^2 n)$	$O(\sqrt{n} \log^2 n)$
GMOT'11	"Extending OS'97"	$O(n^\delta)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
SCSL'11	"Binary Tree"	$O(1)$	$O(1)$	$O(n \log n)$	$O(\log^3 n)$	$O(\log^3 n)$
GMOT'12	"Cuckoo+"	$O(n^\delta)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
SSS'12	"Parallel Buffers "	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$	$O(\log^2 n)$
You?	"Optimal"	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$

= covered in this talk

- Omitted insecure schemes
  - Notably, Pinkas-Reinman (Crypto'10)
- All of these are extensions of G'87 and O'90, except SCSL'11 and SSS'12
- No optimal construction known
- SSS'12 claims to be most practical, despite bad asymptotics

Outline

1. Goldreich's "Square Root" ORAM & Extensions
2. Ostrovsky's "Hierarchical" ORAM & Extensions
3. Cuckoo Hashing ORAMs: Scheme and Attack
4. A "Practical ORAM"

Outline

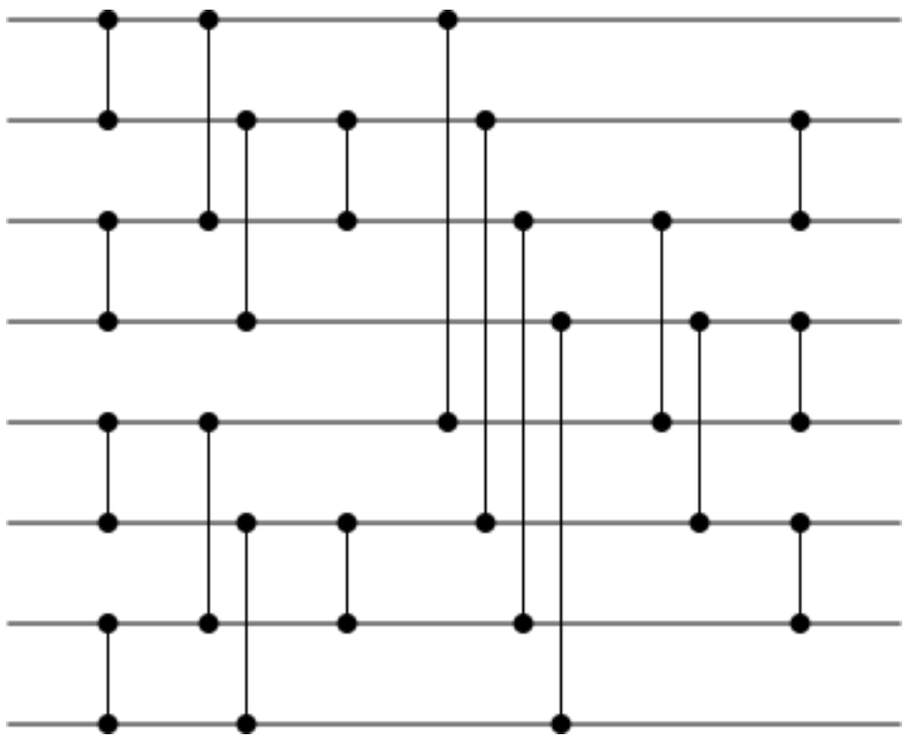
1. **Goldreich's "Square Root" ORAM & Extensions**
2. Ostrovsky's "Hierarchical" ORAM & Extensions
3. Cuckoo Hashing ORAMs: Scheme and Attack
4. A "Practical ORAM"

Basic Tool: Oblivious Shuffling

Claim: Given any permutation π on {1 , ... , N}, we can permute the data according to π with a sequence of ops that does not depend on the data or π.

This means we move data at address i to address π(i).

Proof idea: Use an **oblivious sorting algorithm**. For each comparison in the sort, read both positions and rewrite them, either swapping the data or not (depending on if π(i) > π(j)).



Basic Tool: Oblivious Shuffling

Claim: Given any permutation π on {1 , ... , N}, we can permute the data according to π with a sequence of ops that does not depend on the data or π.

This means we move data at address i to address π(i).

Proof idea: Use an **oblivious sorting algorithm**. For each comparison in the sort, read both positions and rewrite them, either swapping the data or not (depending on if π(i) > π(j)).

Batcher sorting network: $O(N \log^2 N)$ comparisons, fast

AKS sorting network: O(N log N) comparisons, slow in practice

Randomized Shell sort: O(N log N) comparisons, fast,
sorts w.p. 1 - 1/poly - Concrete security loss?

Basic Tool: Oblivious Shuffling

Claim: Given any permutation π on {1 , ... , N}, we can permute the data according to π with a sequence of ops that does not depend on the data or π.

Corollary: Given a key K for a PRP F, we can permute the data according to F(K, ·) using O(1) client memory with a sequence of O(N log N) ops that does not depend on the data or K.

Note: Using O(N) client memory we can do this with O(N) ops by reading everything, permuting locally, and then uploading.

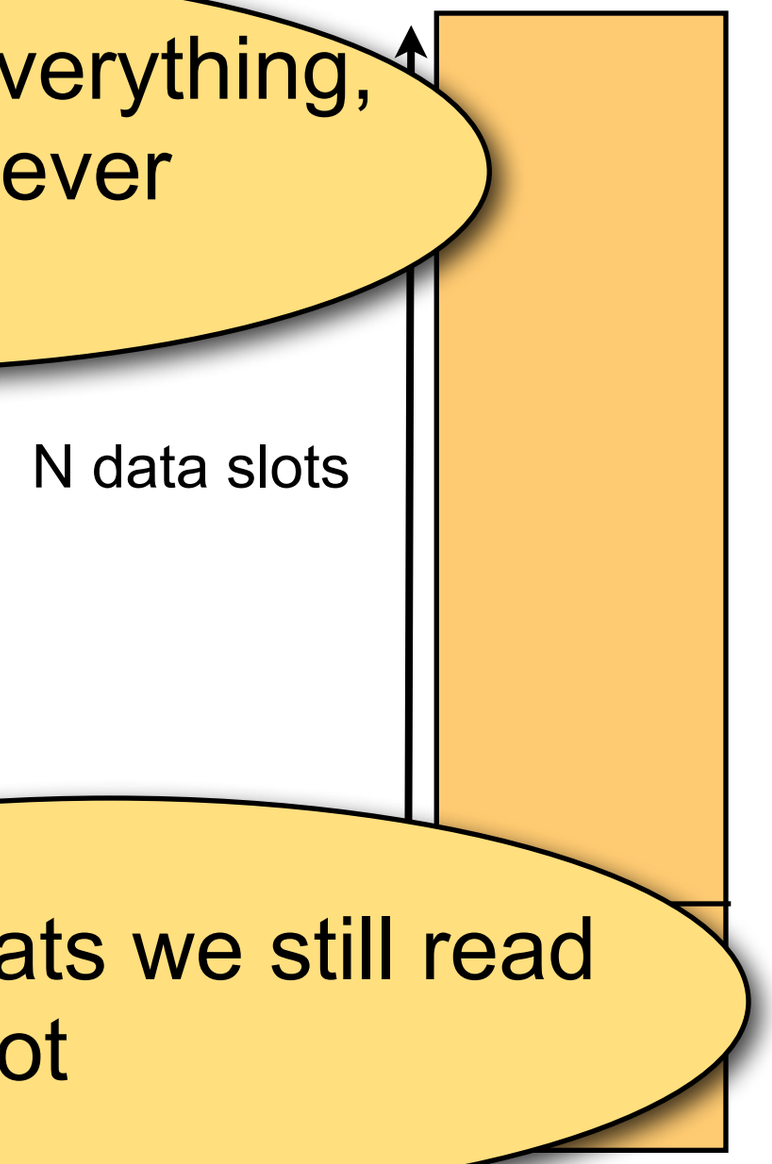
A Simple ORAM Using Shuffling

Server storage: N + C

Client storage: C slo

Shuffle hides everything, assuming we never repeat a read.

Initialization: Pick PRP key. Use it to obliviously shuffle N data slots together with C "dummy" slots.



To read/write a slot:

- If data **is not** in client stor read it from DB
- If data **is** in client storage, read next dummy slot
- Write data into client storage

On repeats we still read a new slot

A Simple ORAM Using Shuffling

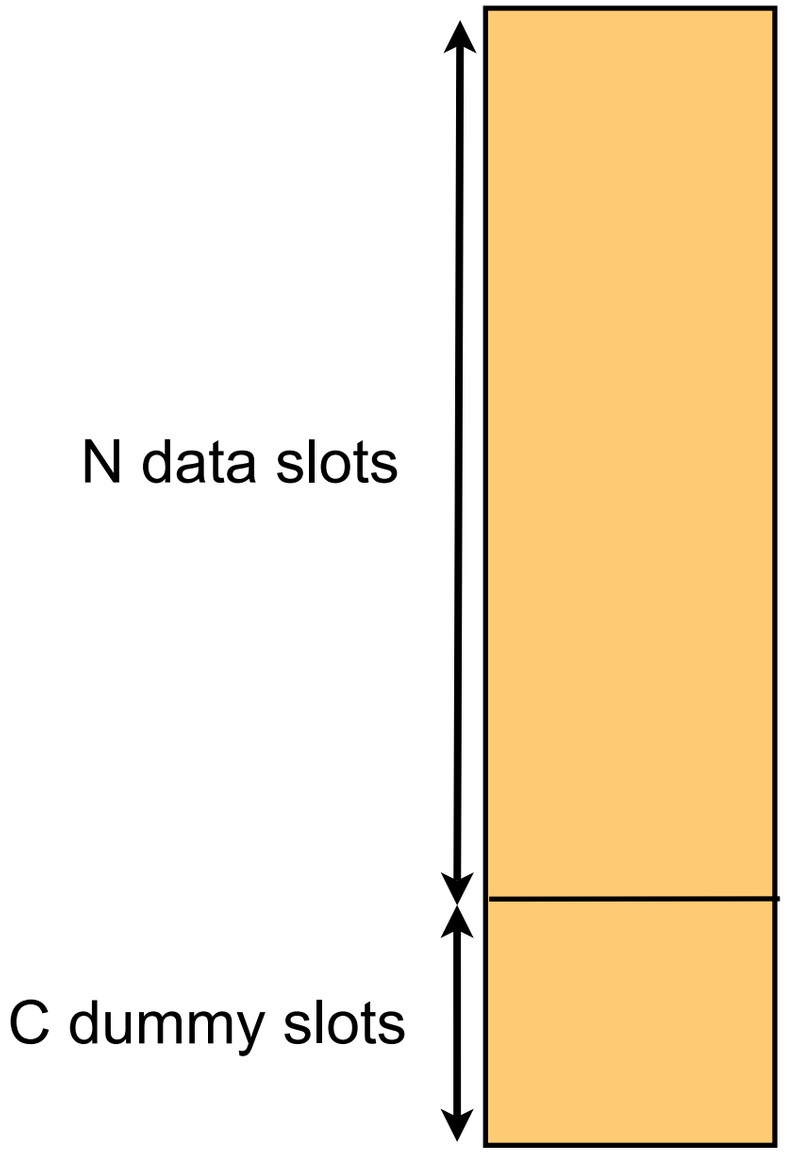
Server storage: N + C slots

Client storage: C slots

After C ops, cache may be full or we may run out of dummy slots.

⇒ Reshuffle and flush cache after every C reads.

Pick new PRF key and shuffle, overwriting “stale” data (i.e. slots that were changed in client cache)



Analyzing the Shuffling ORAM

Security: Relatively easy to prove.

↳ Server sees an oblivious sort and then C unique, random-looking read/writes before reinitializing.

Performance:

Client storage: C slots

Server Storage: N + C slots

Amortized overhead: 1 + (N+C) log(N+C) / C

Worst-case overhead: 1 + (N+C) log(N+C)

One op per read plus cost to shuffle after C reads.

Goldreich's ORAM

Basic observation: We can just put the cache on the DB and read it back each time.

Server storage: N + 2C slots

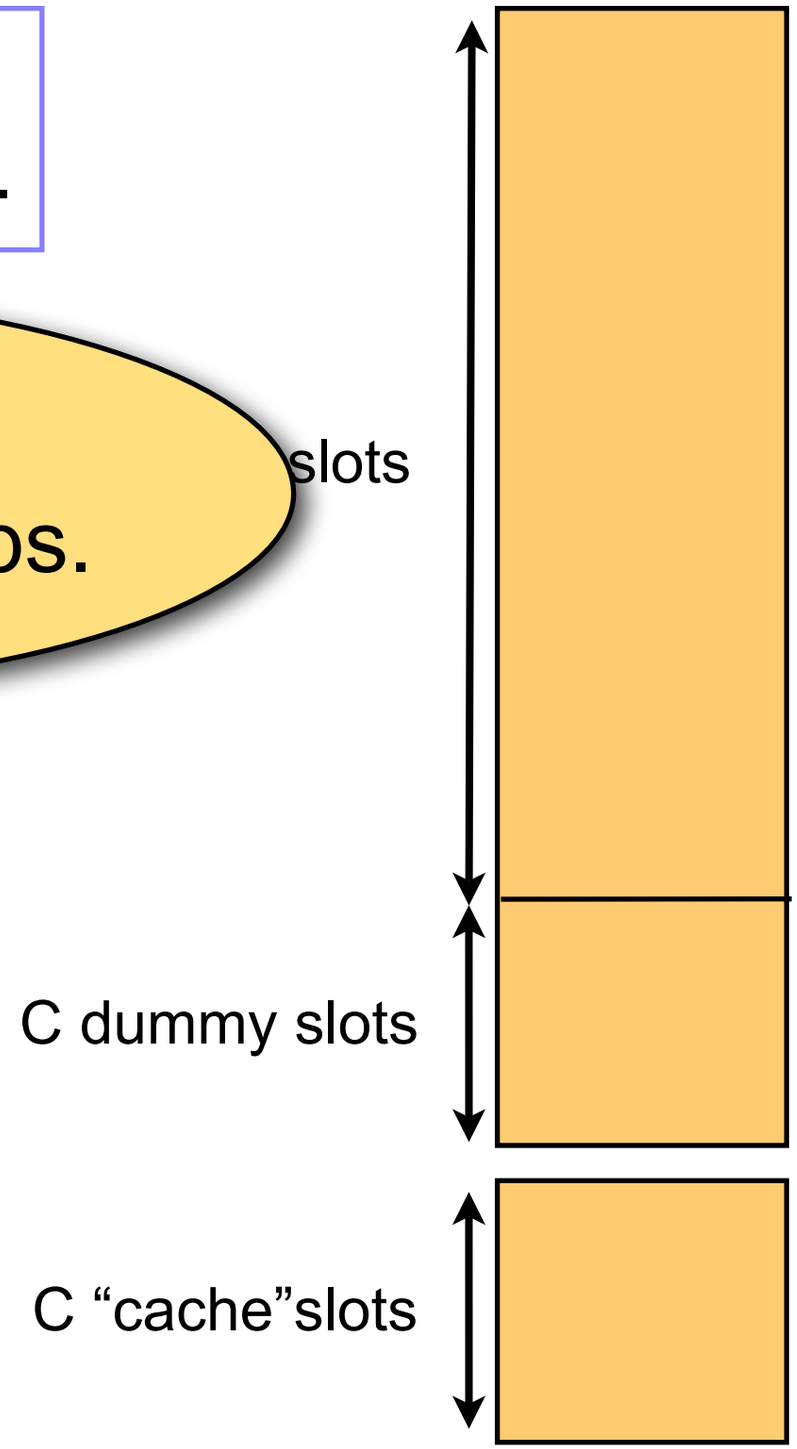
Client storage: C slots

Initialization: Sam

Use same shuffling procedure after C ops.

To read a slot:

- Scan cache from server
- If data is not in server cache, read it from main memory
- If data is in server cache, read next dummy slot
- Write data into server cache



Performance of Goldreich's ORAM

	Client Memory	Client Storage	Server Storage	Amortized Cost	Worst-case Cost
#1	O(1)	C	N + C	$(C + (N+C)\log(N+C))/C$	$1 + (N+C)\log(N+C)$
#2	O(1)	O(1)	N + 2C	$(C^2 + (N+C)\log(N+C))/C$	$C + (N+C)\log(N+C)$

Take $C = N^{1/2}$:

	Client Memory	Client Storage	Server Storage	Amortized Cost	Worst-Case Cost
#1	O(1)	**$N^{1/2}$**	O(N)	$O(N^{1/2} \log N)$	O(N log N)
#2	O(1)	**O(1)**	O(N)	$O(N^{1/2} \log N)$	O(N log N)

Batcher sort ⇒ extra log N factor in costs

De-Amortizing Goldreich's ORAM

[Ostrovsky, Shoup'97]

Observation: For our oblivious sorts, all comparisons are predetermined, so the work can be divided up and done in small bursts instead of one big sort.

- This doesn't immediately work for de-amortizing because we still need to do reads/writes in between the bursts.

- Instead:
  - Maintain two copies of the database, one "old" and one "current".
  - Sort the old one in bursts while using the current one
  - After finishing the sort, swap the copies.

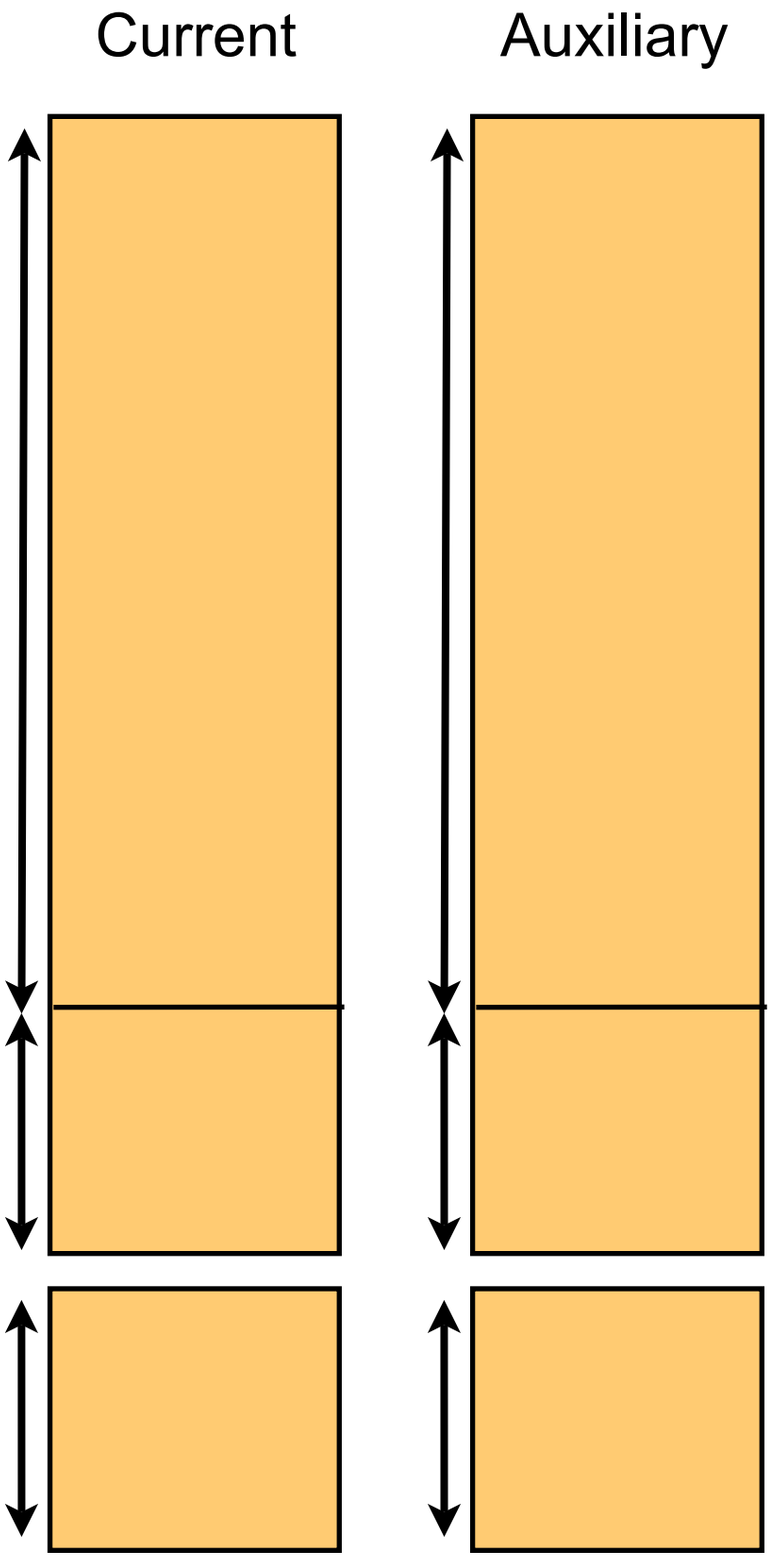
De-Amortizing Goldreich's ORAM

Initialization: Same, except allocate and two tables and caches.

To read a slot:

- Read from Current as before, update Current's cache
  - Also check Auxiliary's cache
- Then perform next chunk of shuffle operations on Auxiliary
- After $N^{1/2}$ ops, "swap" Current and Auxiliary

- Correct simulation because all of the changed slots will be in aux cache



Outline

1. Goldreich's "Square Root" ORAM & Extensions
2. **Ostrovsky's "Hierarchical" ORAM & Extensions**
3. Cuckoo Hashing ORAMs: Scheme and Attack
4. A "Practical ORAM"

Ostrovsky's ORAM

- Much more complicated technique for hiding repeated access to same slots
- Reduces amortized cost from $O(N^{1/2} \log N)$ to $O(\log^3 N)$
- Requires fancier slides

Ostrovsky's ORAM: Storage Layout

Server storage:

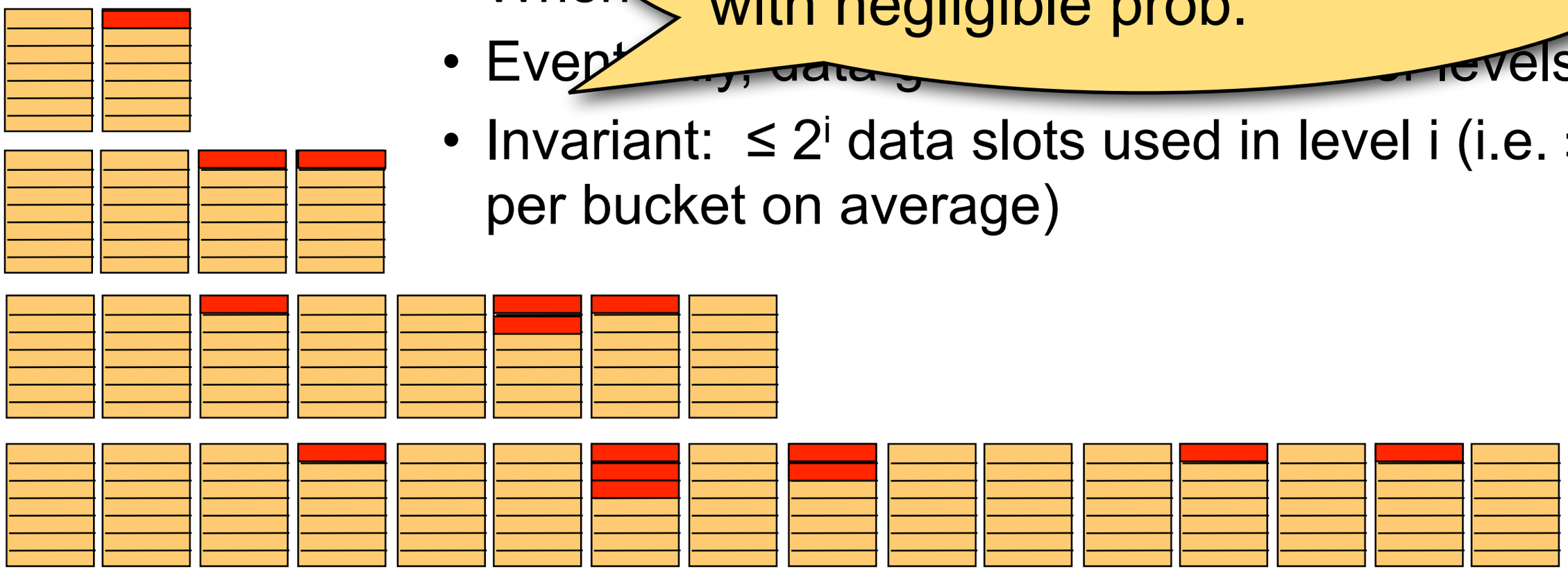
- log N "levels"
- Level i contains 2^i buckets
- Buckets each contain log N slots

Client storage:

- PRF key K_i for each level

- Data
- When
- Even
- Invariant: ≤ 2^i data slots used in level i (i.e. ≤ 1 per bucket on average)

Assuming data is randomly put into buckets, overflow happens with negligible prob.

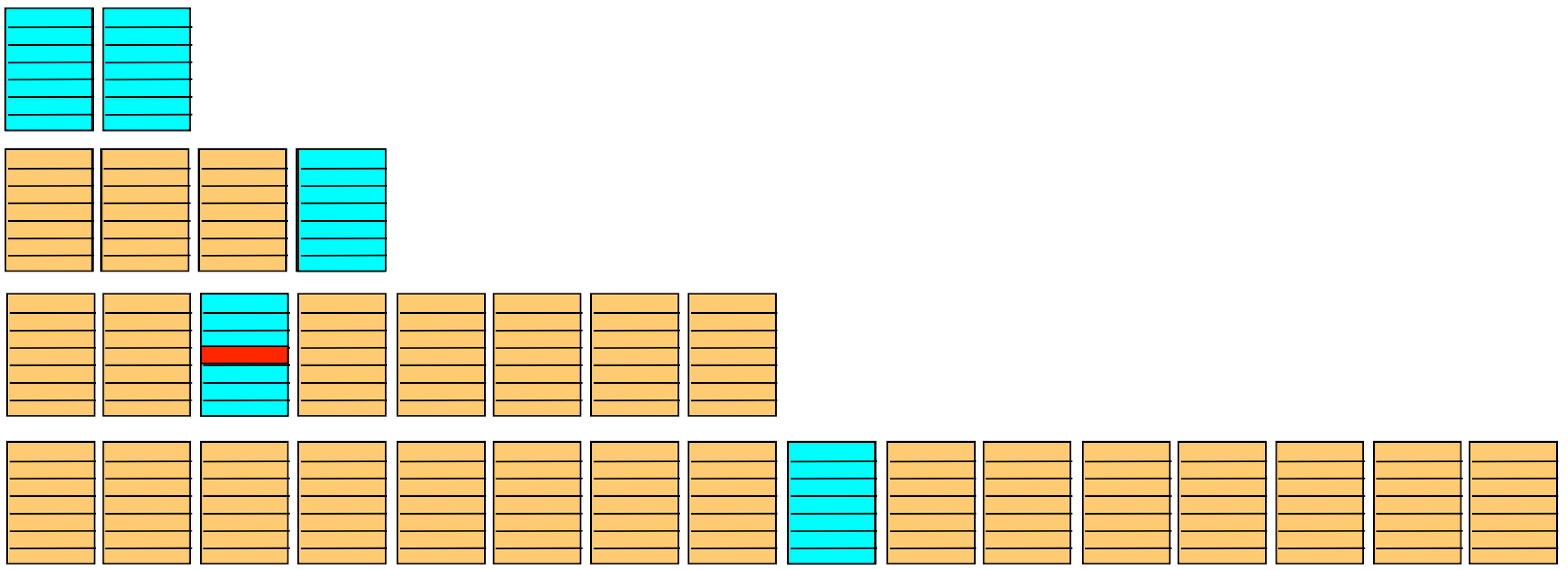


Ostrovsky's ORAM: Read/Write Op

Read/Write(addr)

- Scan both top buckets for data
- At each lower level, scan exactly one bucket
  - **Until found**, scan bucket at $F(K_i, addr)$ on that level
  - **After found**, scan a random bucket on that level
- Write data into bucket $F(K_1, addr)$ on level 1
- Perform a "shuffling procedure" to maintain invariant

Example of Read/Write



Computation during Read/Write(red address):

1. Scan both buckets at level 1
2. Scan bucket $F(K_2, addr) = 4$ in level 2
3. Scan $F(K_3, addr) = 3$ in level 3 (finding data)
4. Scan a random bucket in level 4
5. Move found data to level 1

Shuffling Procedure

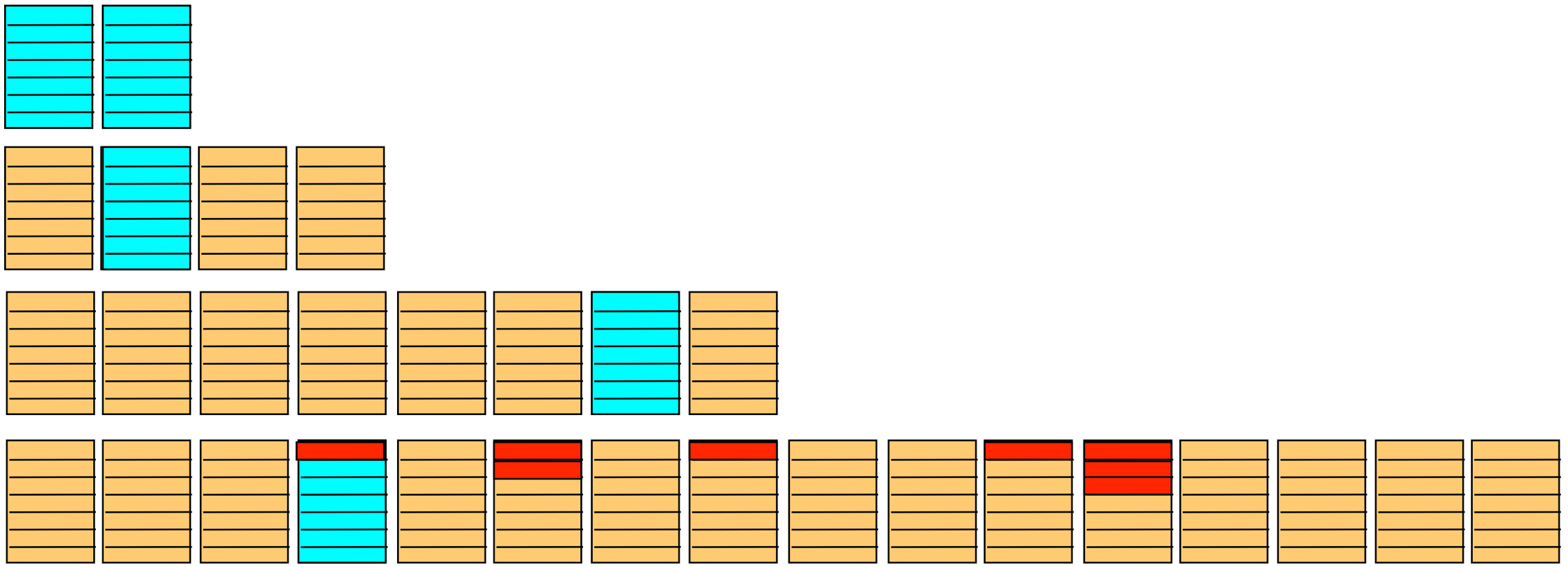
- We “merge levels” so that each level has ≤ 1 slot per bucket on average

After T operations:

- Let D = {max x: 2^x divides T}
- For i=1 to D
  - Pick PRF key for level i+1
  - Shuffle data in levels i and i+1 together into level i+1 using new key

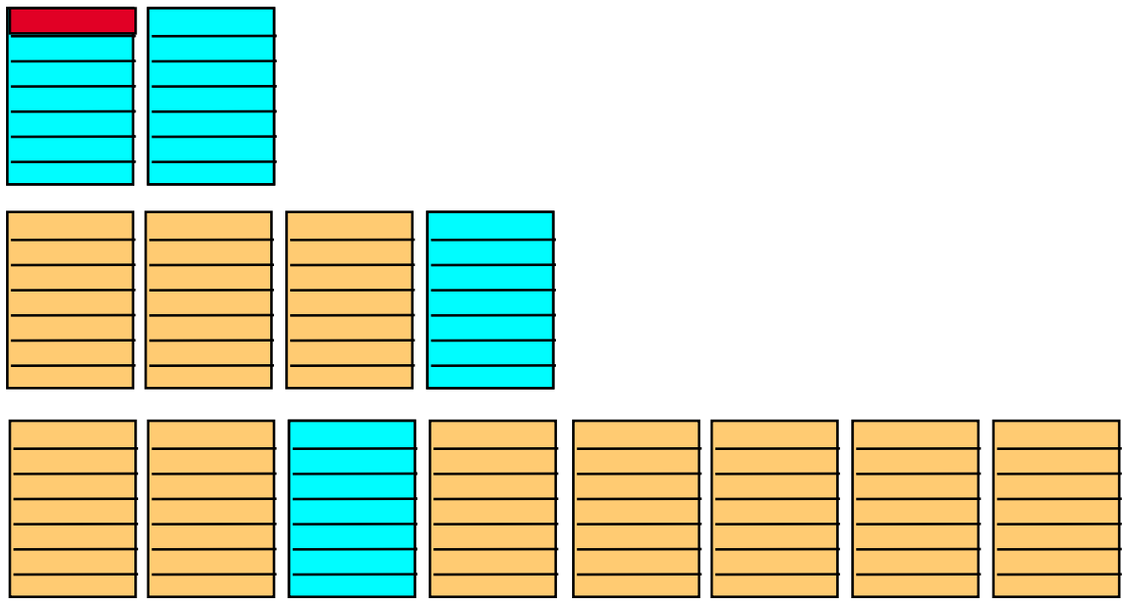
- Level i is shuffled after every 2^i ops.

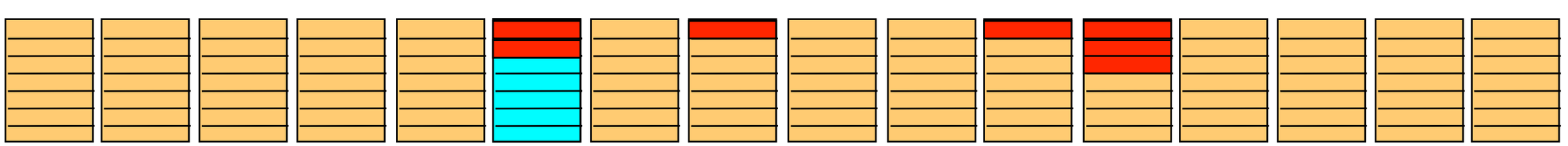
Example: Read/Writes with Shuffling



1. Read a slot

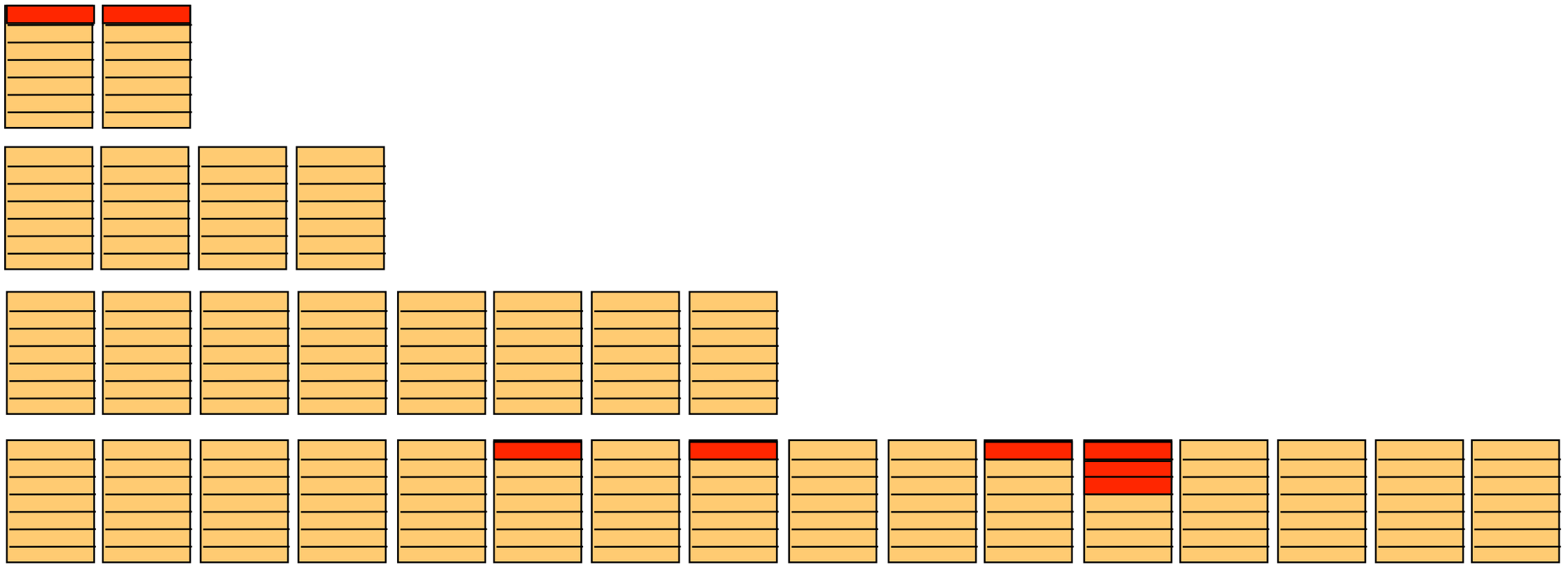
Example: Read/Writes with Shuffling





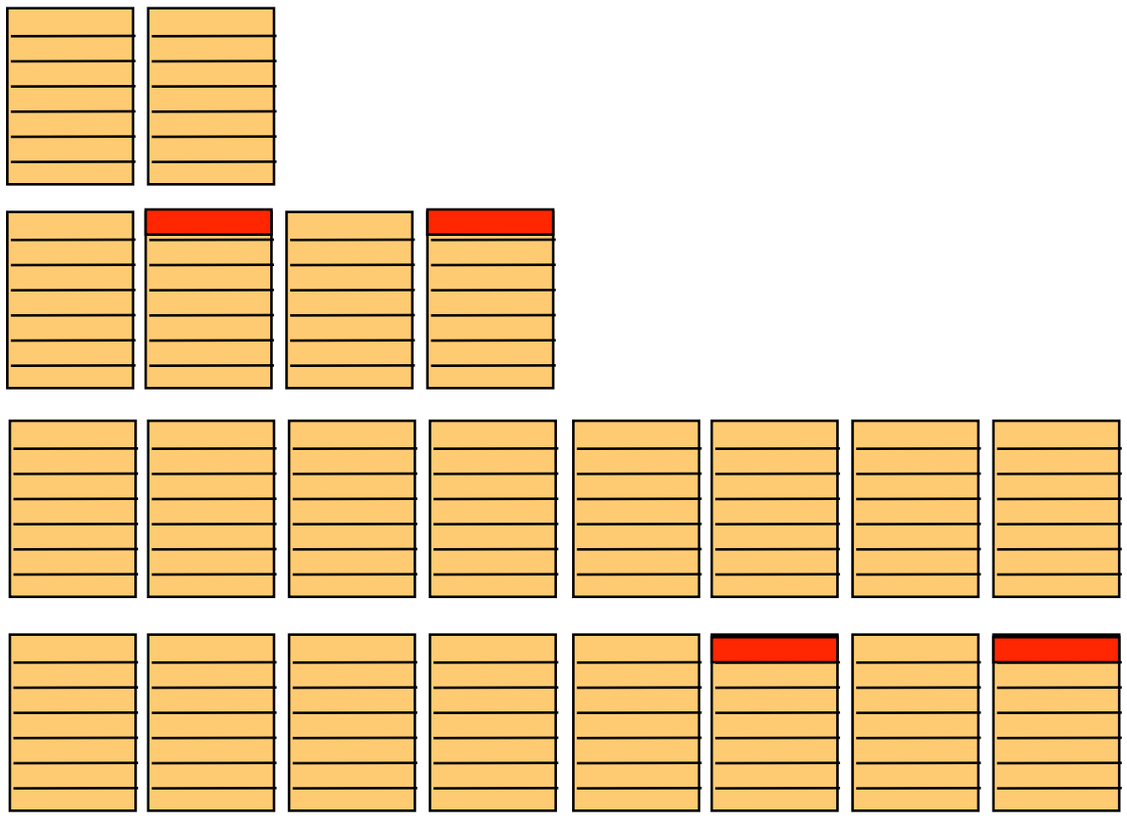
1. Read a slot
2. Read another slot

Example: Read/Writes with Shuffling



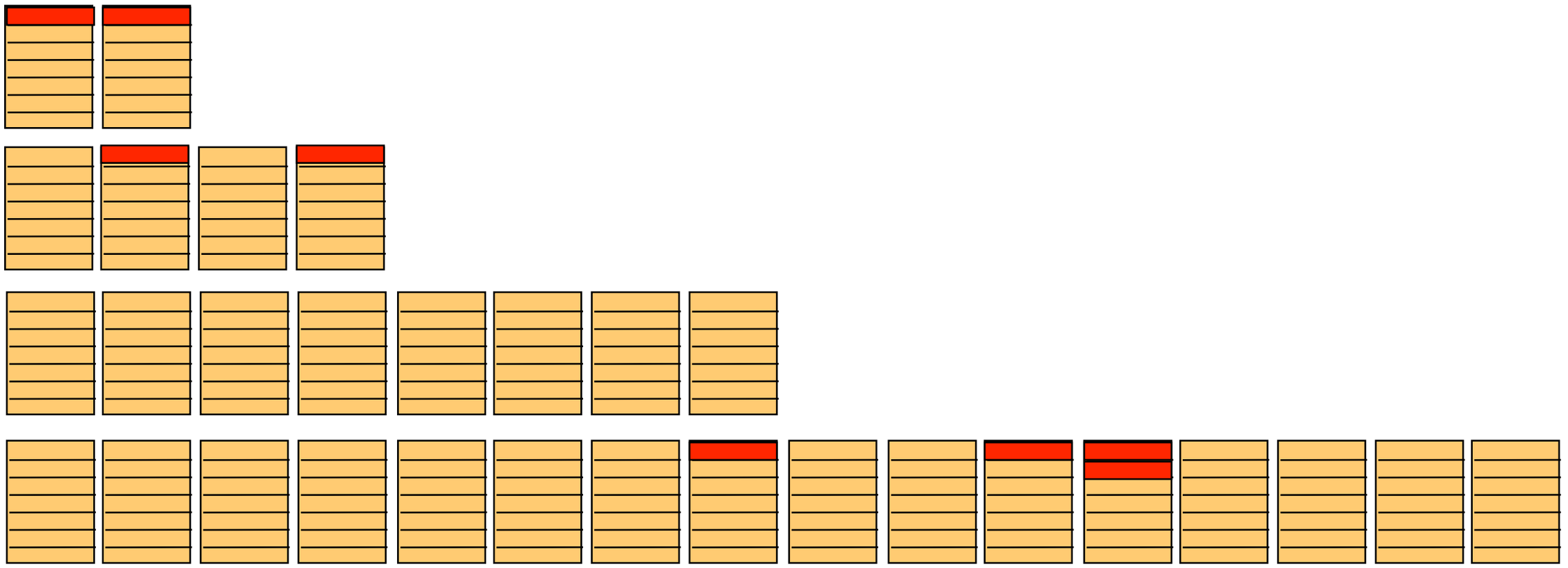
1. Read a slot
2. Read another slot
   - Level 1 shuffled after $2 = 2^1$ ops (stops there)

Example: Read/Writes with Shuffling



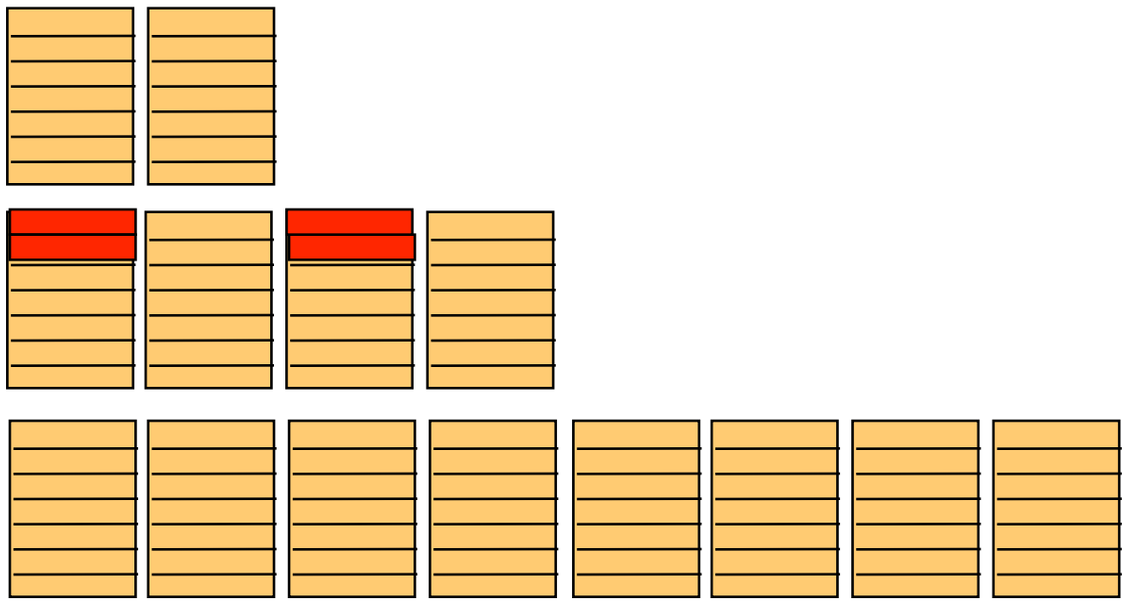
1. Read a slot
2. Read another slot
   - Level 1 shuffled after 2 = 2^1 ops (stops there)
3. Two more reads

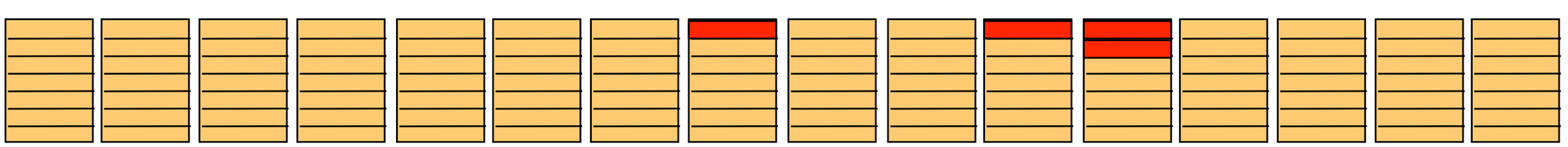
Example: Read/Writes with Shuffling



1. Read a slot
2. Read another slot
   - Level 1 shuffled after 2 = 2^1 ops (stops there)
3. Two more reads
   - Level 1 shuffled after 2 = 2^2 ops

Example: Read/Writes with Shuffling





1. Read a slot
2. Read another slot
   - Level 1 shuffled after 2 = 2^1 ops (stops there)
3. Two more reads
   - Level 1 shuffled after 4 = 2^2 ops
   - Level 2 shuffled after 4 = 2^2 ops (stops there)

Security of Ostrovsky's ORAM

Security proof is more delicate than the first one.

Key observation: This scheme never uses the value $F(K_i, addr)$ on the same (key, address) twice.

↳ **Why?** Suppose client touches for the same address twice.

- After the first read, data is promoted to level 1.
- During the next read:
  - If it is still on level 1, then we don't evaluate F at all.
  - If it is has been moved, a new key must have been chosen for that level since last read due to shuffling.

Using key observation, all reads look like random bucket scans.

Ostrovsky's ORAM: Performance

Worst-case overhead: O(N $\log^3$ N)

- Shuffling level i takes O(2^i*i*log (N)) comparisons
- In worst case shuffle all levels, costing:

$$\Sigma_{i=0}^{\log N} O(2^i i \log N) = \Sigma_{i=0}^{\log N} O(N \log^2 N) = O(N \log^3 N)$$

Average-case overhead: O($\log^3$ N)

- Shuffling level i is amortized over 2^i operations
- Amortized work for all (log N) levels:

$$\Sigma_{i=0}^{\log N} O(i \log N) = \Sigma_{i=0}^{\log N} O(\log^2 N) = O(\log^3 N)$$

Storage: O(N log N) slots

Extensions of Ostrovsky's ORAM

De-amortized variant: Can shuffle incrementally as before.

[Ostrovsky, Shoup'97]

↳ Gives $O(\log^3 N)$ worst-case overhead, doubles server storage

More advanced sorting: Use $O(N^{\delta})$ client storage when sorting to save log N factor in communication.

[Williams, Sion, Sotakova'08]

↳ $(\log^2 N)$ amortized cost.

Outline

1. Goldreich's "Square Root" ORAM & Extensions
2. Ostrovsky's "Hierarchical" ORAM & Extensions
3. **Cuckoo Hashing ORAMs: Scheme and Attack**
4. A "Practical ORAM"

Improved Performance via Cuckoos

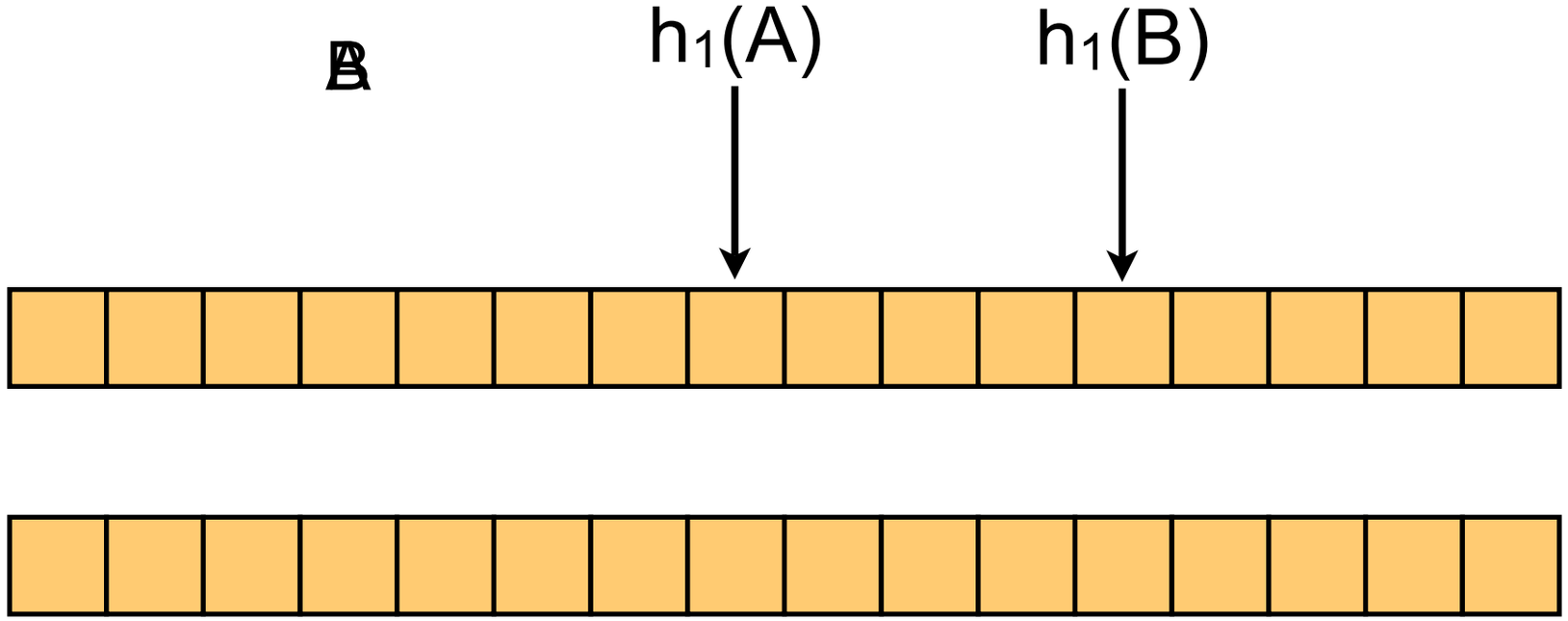
- Replace bucket-lists with more efficient hash table

	Storage	Amortized Overhead
Ostrovksky'90	O(N log N)	O(N $\log^3$ N)
Pinkas-Reinman'10	O(N)	O(N $\log^2$ N)

Cuckoo Hashing

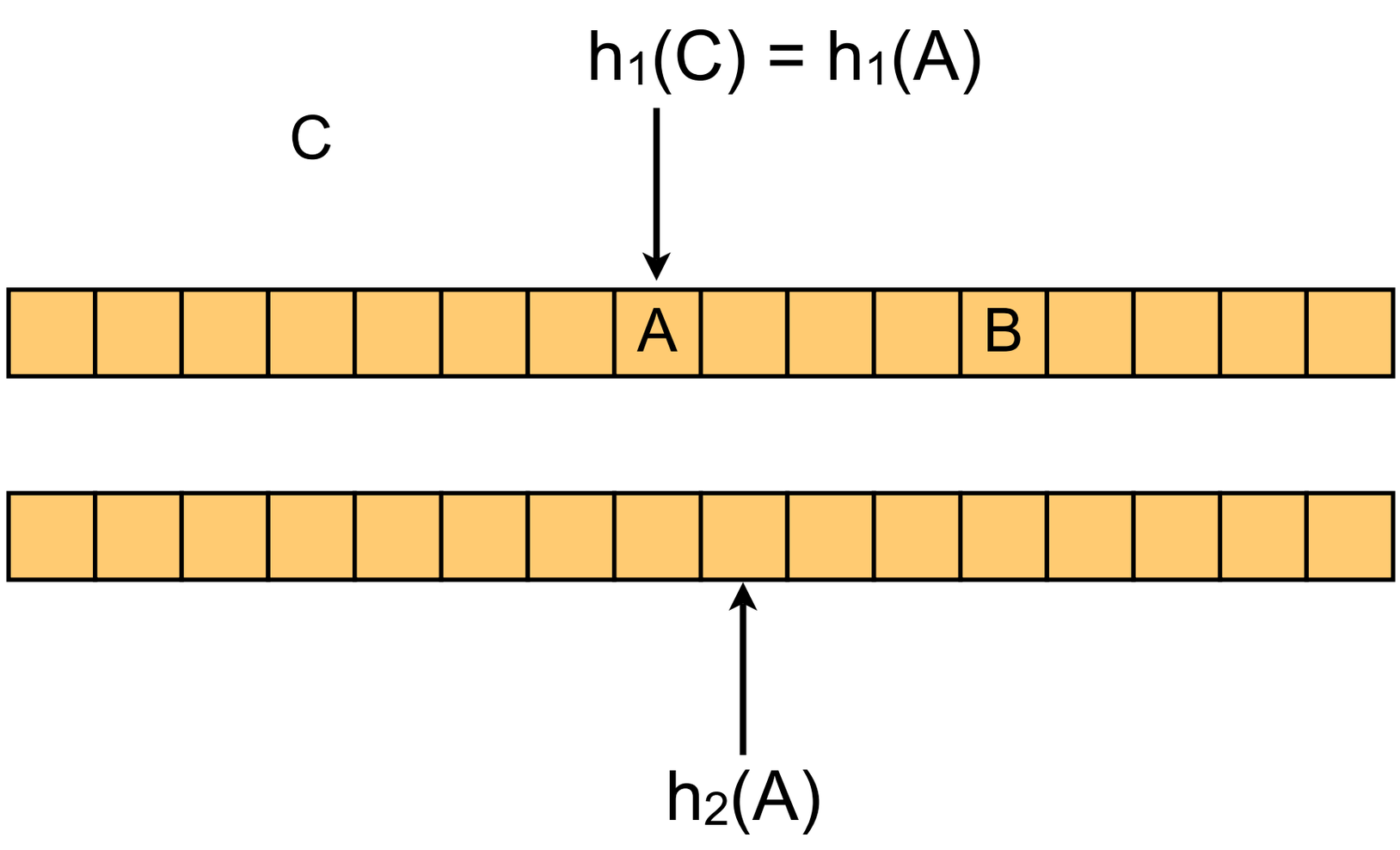
Cuckoo Hashing

- Uses two tables of size n.
- Pick two hash functions (h_1, h_2) mapping data into {1, …, n}.
- Data x is stored at either $h_1(x)$ in table 1 or $h_2(x)$ in table 2.



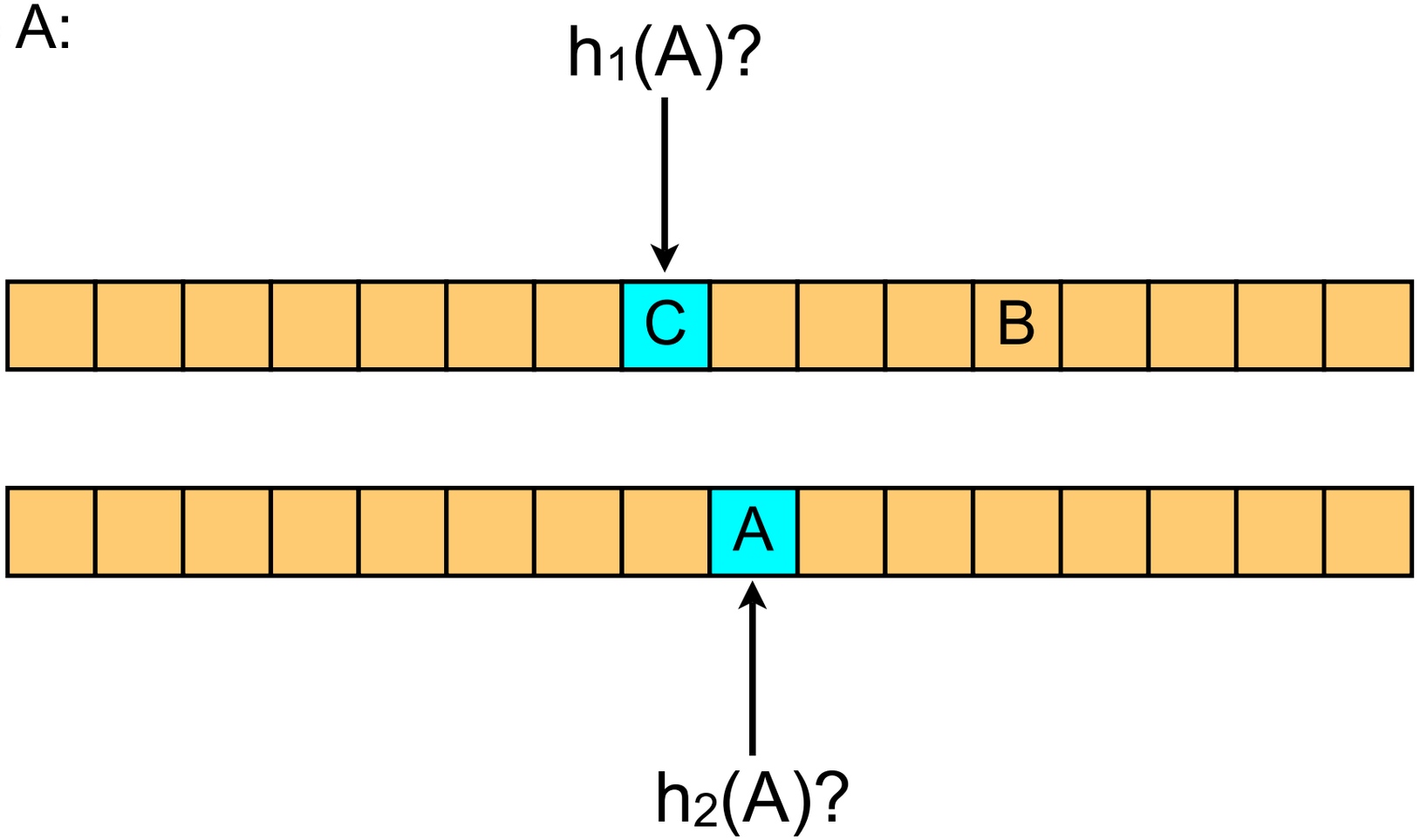
Cuckoo Hashing

- Uses two tables of size n.
- Pick two hash functions (h_1, h_2) mapping data into {1, …, n}.
- Data x is stored at either $h_1(x)$ in table 1 or $h_2(x)$ in table 2.

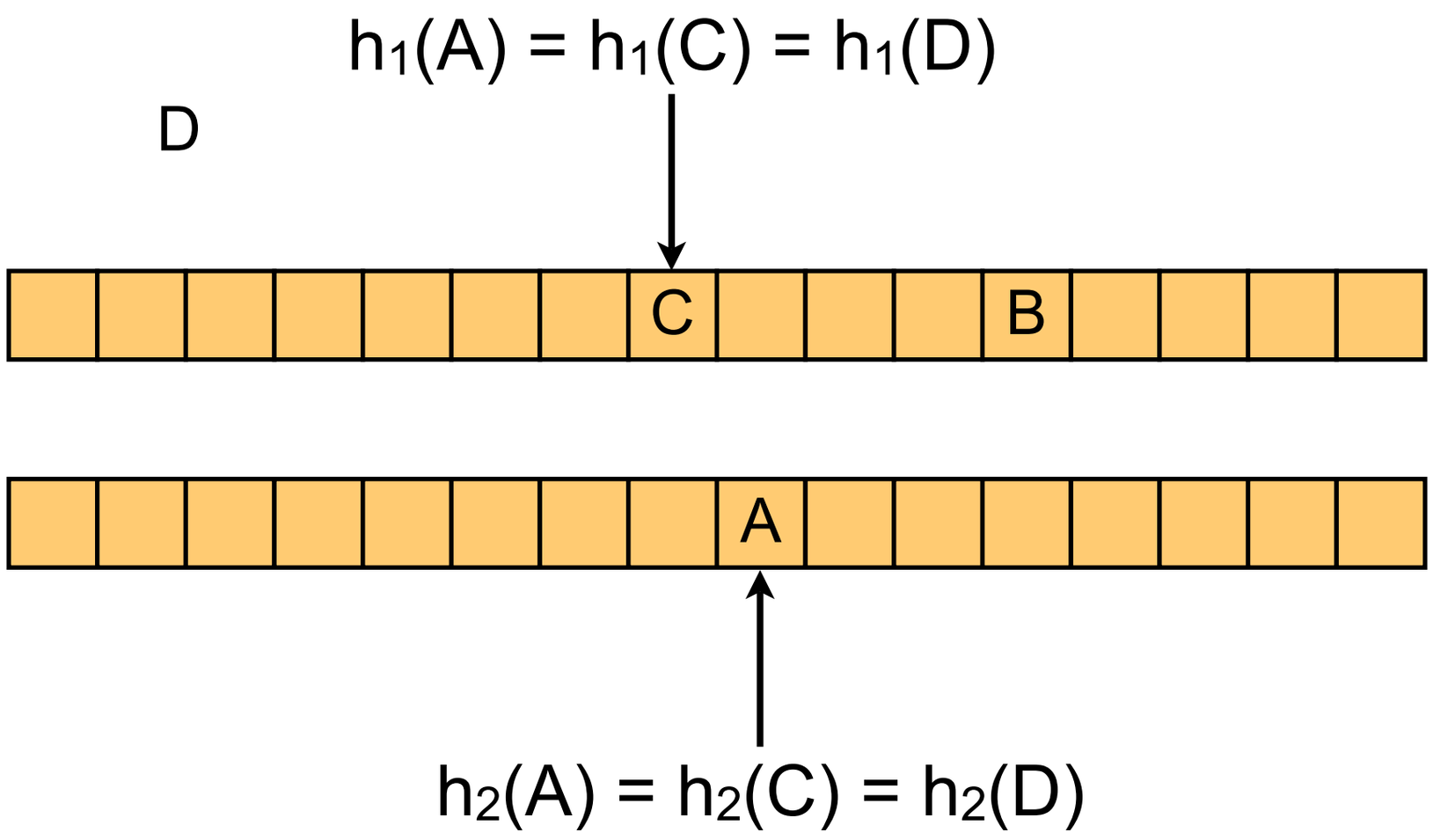


Look-up is constant-time.

To look up A:



Failures occur when x items hash to same (x-1) slots in both tables.



In practice, we abort the insertion after a chain of c log n evictions.

Theorem (Pagh-Rodler'01): After (1-ϵ)n insertions probability of failure is $\Theta(1/n^2)$, where ϵ is constant.

Pinkas-Reinman ORAM

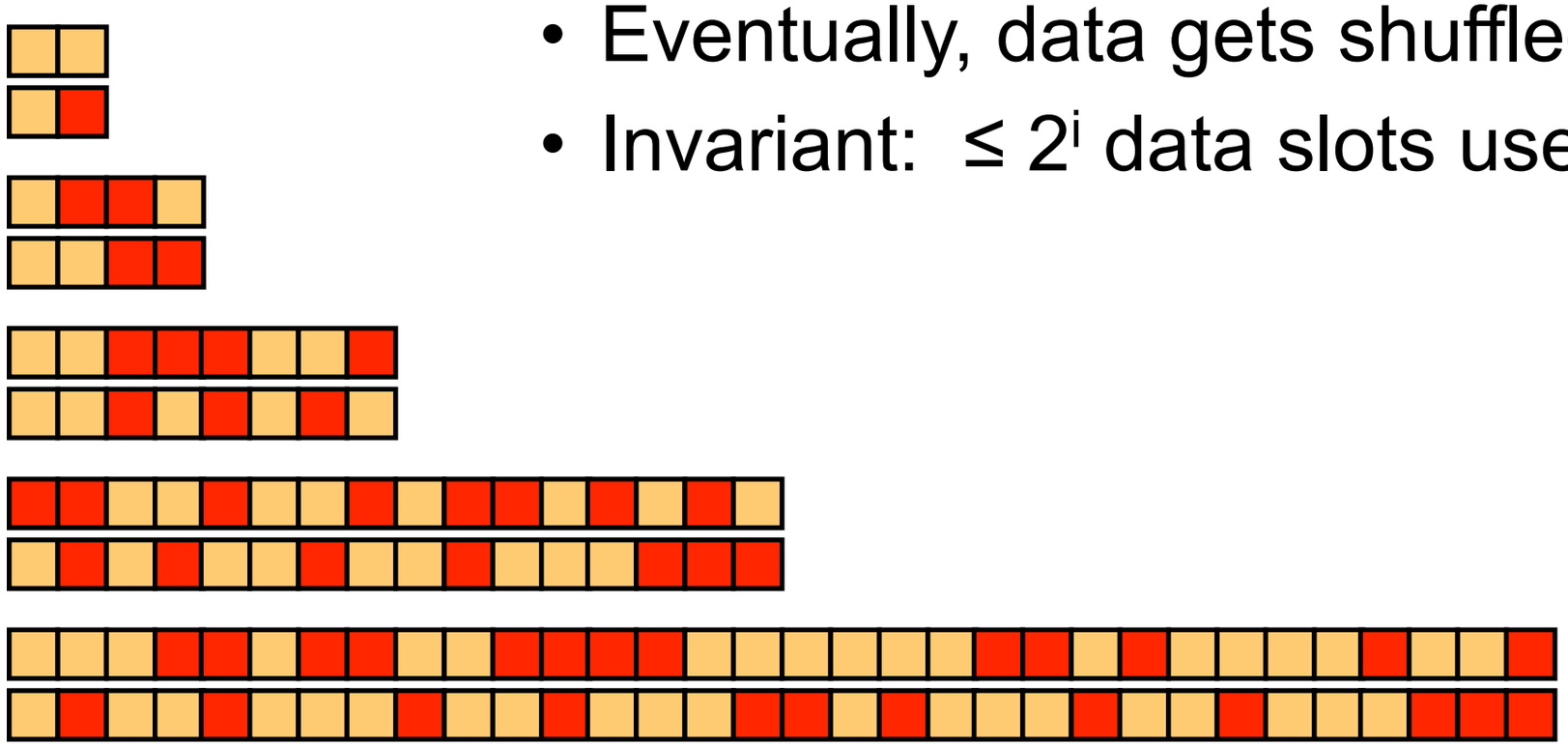
Server storage:

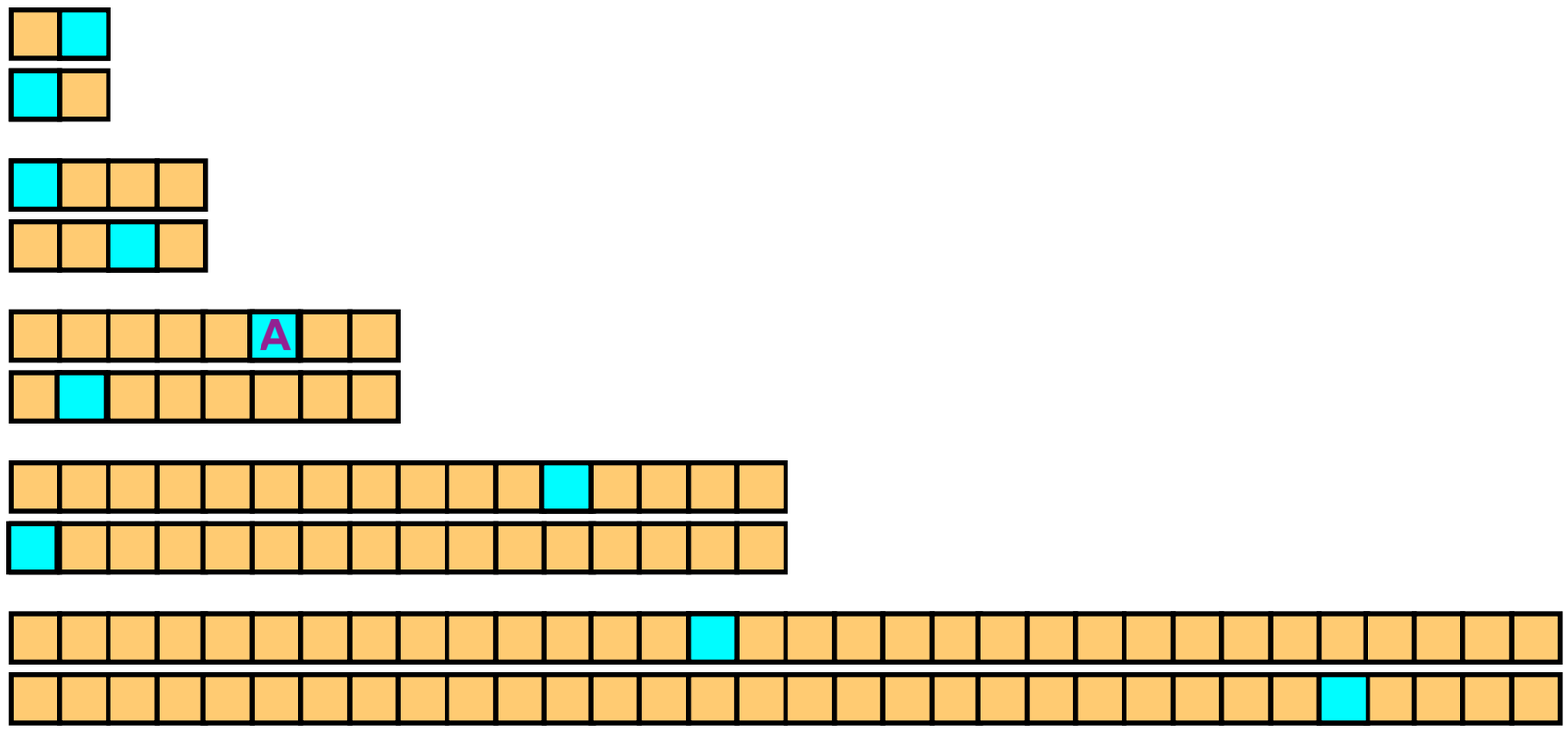
- log N “levels”
- Level i is cuckoo hash for 2^i slots

Client storage:

- Hash functions ($h_{i,1}$ $h_{i,2}$) for each level

- When accessed, data gets moved to level 1
- Eventually, data gets shuffled to lower levels
- Invariant: ≤ 2^i data slots used in level i



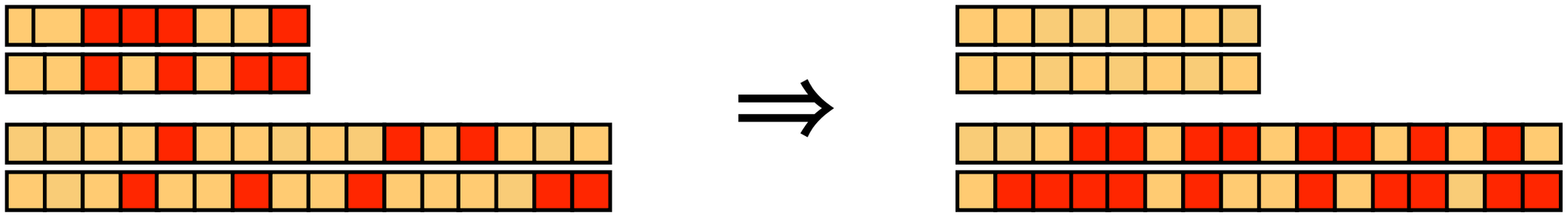


Computation during Read/Write(A):

1. Read slots $h_{1,1}(A)$, $h_{1,2}(A)$ at level 1
2. Read slots $h_{2,1}(A)$, $h_{2,2}(A)$ at level 2
3. Read slots $h_{3,1}(A)$, $h_{3,2}(A)$ at level 3 (found A)
4. Read two random slots in level 4 & 5
5. Insert A into level 1 (evicting data, etc)

Shuffling Procedure

- After 2^i operations, rehash level i together with level i+1 to prevent overflows



- Oblivious rehashing can be done with O(1) oblivious sorts
- Cuckoo hash will fail with probability ~ $1/n^2$
  - PR'10 picks new hash functions until no failure

Pinkas-Reinman ORAM: Performance

Worst-case overhead: O(N $\log^2$ N)

- Level i has ≈2*2^i slots: takes O(2^i*log 2^i) = O(2^i*i) ops to shuffle/rehash
- Worst case, shuffle all levels: $\sum_{i=1}^{\log N} O(2^i i) = \sum_{i=1}^{\log N} O(N \log N) = O(N \log^2 N)$

Average-case overhead: O($\log^2$ N)

- Rehash of level i is amortized over 2^i ops
- Total amortized overhead: $\sum_{i=1}^{\log N} O(2^i i/2^i) = \sum_{i=1}^{\log N} O(\log N) = O(\log^2 N)$

Storage: O(N) slots

- Because $\sum_{i=1}^{\log N} 2(1+\varepsilon)2^i = O(2^{\log N+1}) = O(N)$

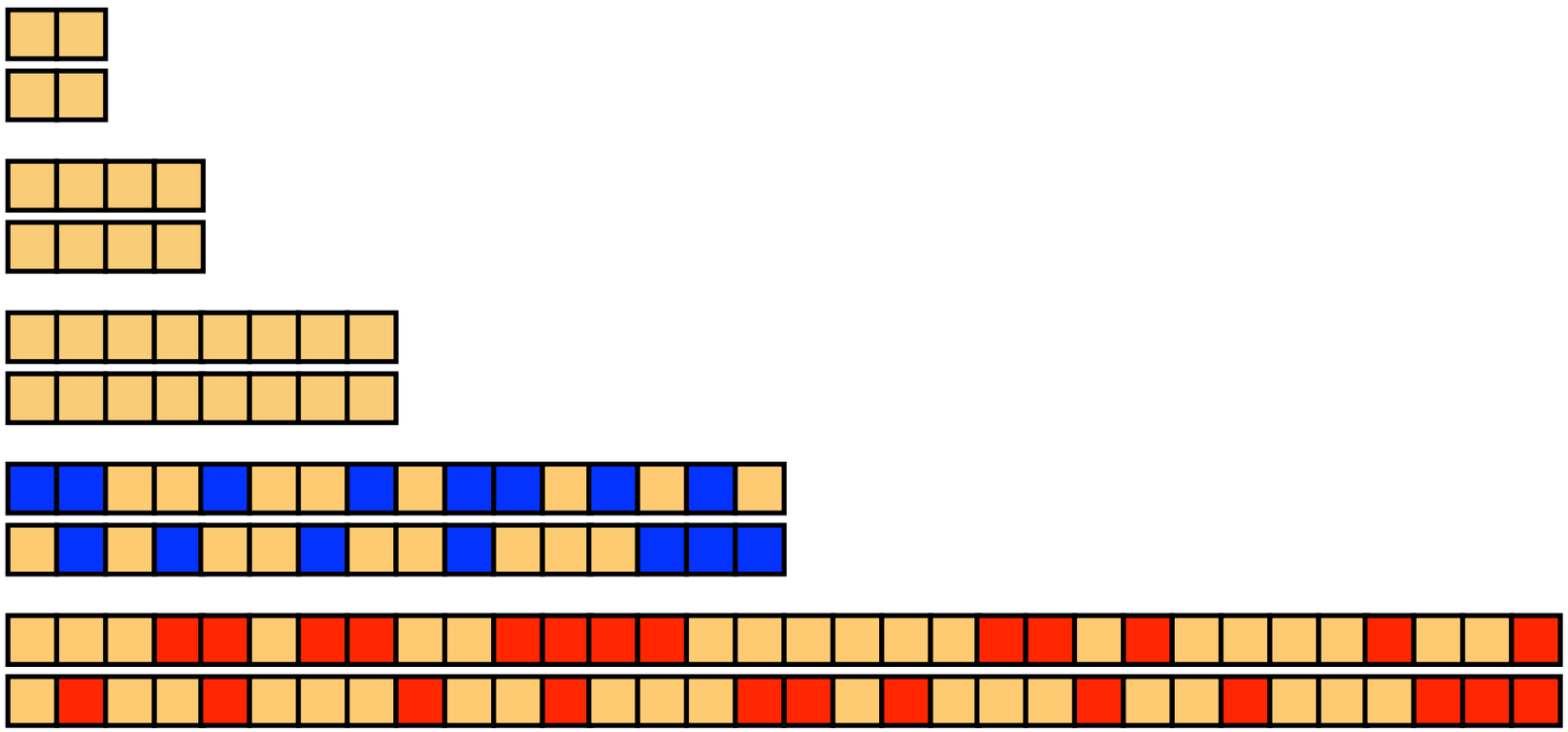
Pinkas-Reinman ORAM: Performance

$n =$ number of slots

$k =$ number of requests

$\log_2 n$	n	$k = n - 10$ (# of req.)	$k \log^2 k$	#operations	ops per request	const of $O(k \log^2 k)$
10	1024	1014	101113	15445582	15232	152
11	2048	2038	246281	38081523	18685	154
12	4096	4086	588038	91975576	22509	156
13	8192	8182	1382383	218482493	26702	158
14	16384	16374	3208900	511882978	31261	159
15	32768	32758	7370117	1185355399	36185	160
16	65536	65526	16774194	2717439532	41471	162
17	131072	131062	37876427	6175479249	47118	163
18	262144	262134	84930896	13926487414	53127	163
19	524288	524278	189263809	31192732955	59496	164
20	1048576	1048566	419425822	69442426048	66226	165

Pinkas-Reinman'10 is not Secure



We define two clients that server can distinguish:

Both clients start with:

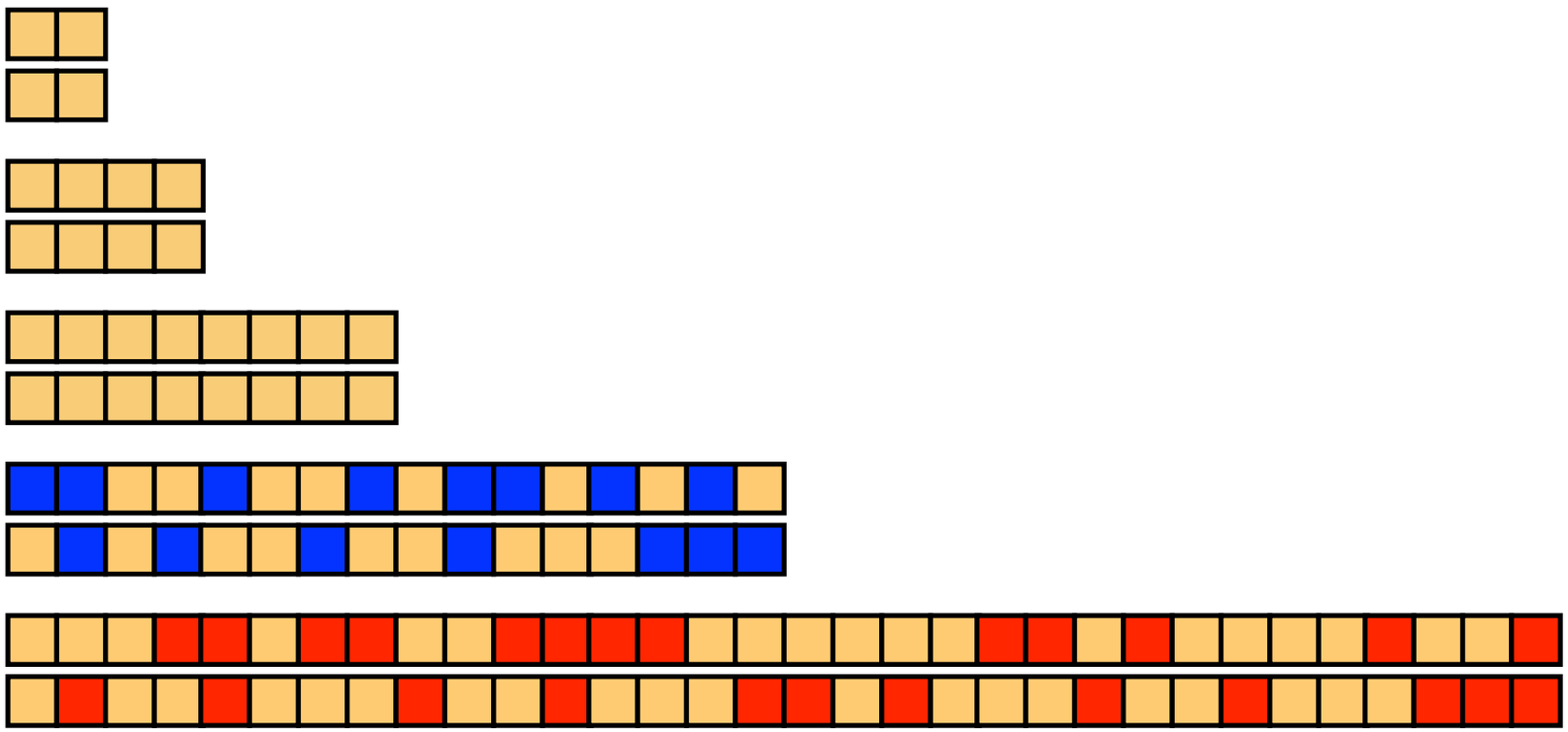
- Query server until blue data is on one level and red data is on next level

Then they differ in one last step:

Client 1: Read several blue slots

Client 2: Read several red slots

Pinkas-Reinman'10 is not Secure



Claim: Server can distinguish clients w/ advantage ~$1/n^6$:

- A query for red slot ⇒ Emulator accesses red slots in last table
- A query for blue slot ⇒ Emulator accesses random slots in last table
- Server can watch for three accesses on last level that touch same pair of buckets (i.e. cause Cuckoo failure)
  - Happens w.p. ~$1/n^6$ if client accesses blue data
  - Happens w.p. 0 if client accesses red data

An Approach to Patching the Problem

- Rehashing the data to avoid failure means that real accesses look different from random access

Observation: If probability of failure were negligible, then PR'10 is secure.

Cuckoo Hashing

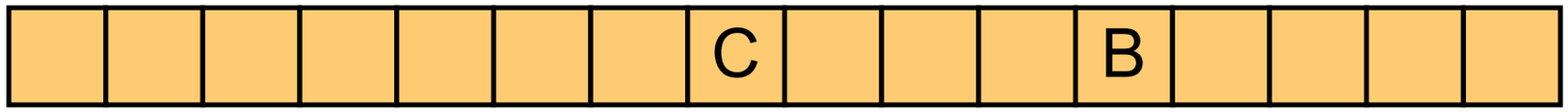
Cuckoo Hashing with (a) Stash

+

Cuckoo Hashing with a Stash

Same tables as before, plus small extra table called a “stash”.

- If inserting an item causes failure, put it in the stash.
- When reading, check stash if item not in main tables.



Let stash size = s.

Theorem (Goodrich-Mitzenmacher’11): After (1-ϵ)n insertions probability of failure is $O(n^{-s})$, where ϵ is constant.

PR'10 + Stashing (GM'11)

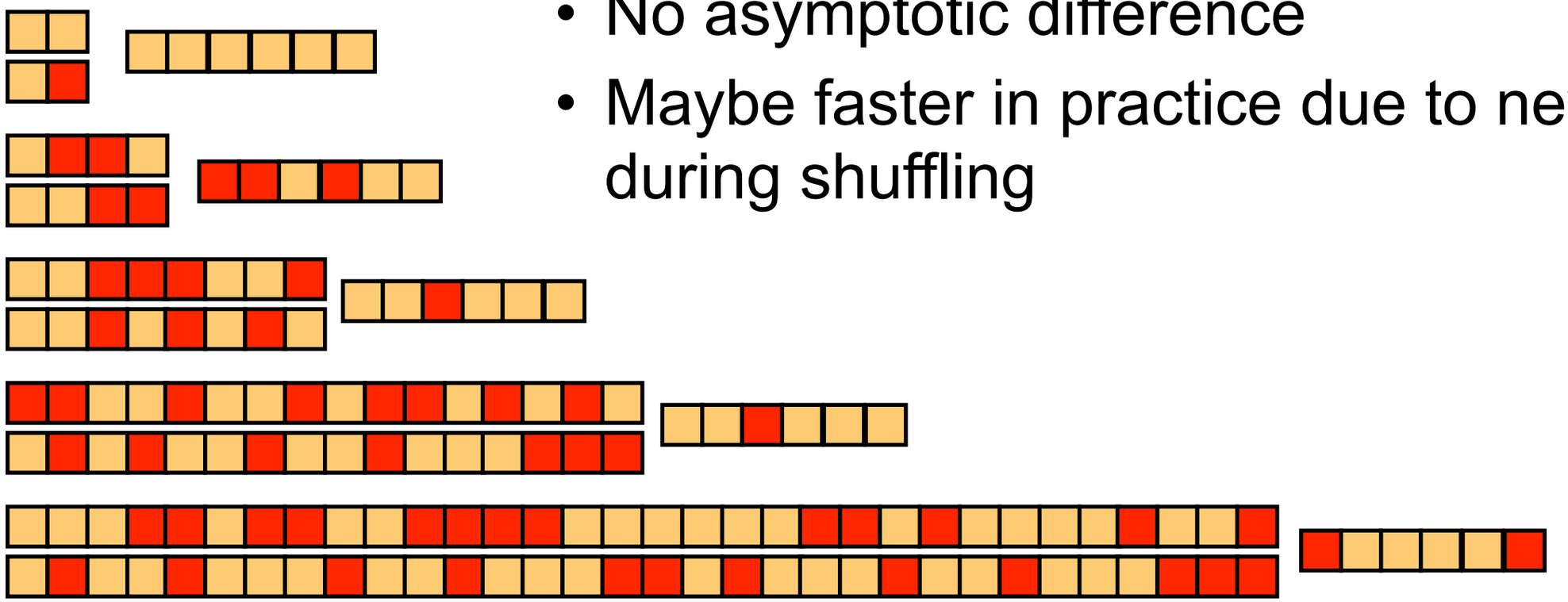
Server storage:

- log N "levels"
- Level i is cuckoo hash table for 2^i slots, and a log N-size stash

Client storage:

- Hash functions ($h_{i,1}$ $h_{i,2}$) for each level

- Accesses must scan all stashes each time
- No asymptotic difference
- Maybe faster in practice due to never failing during shuffling



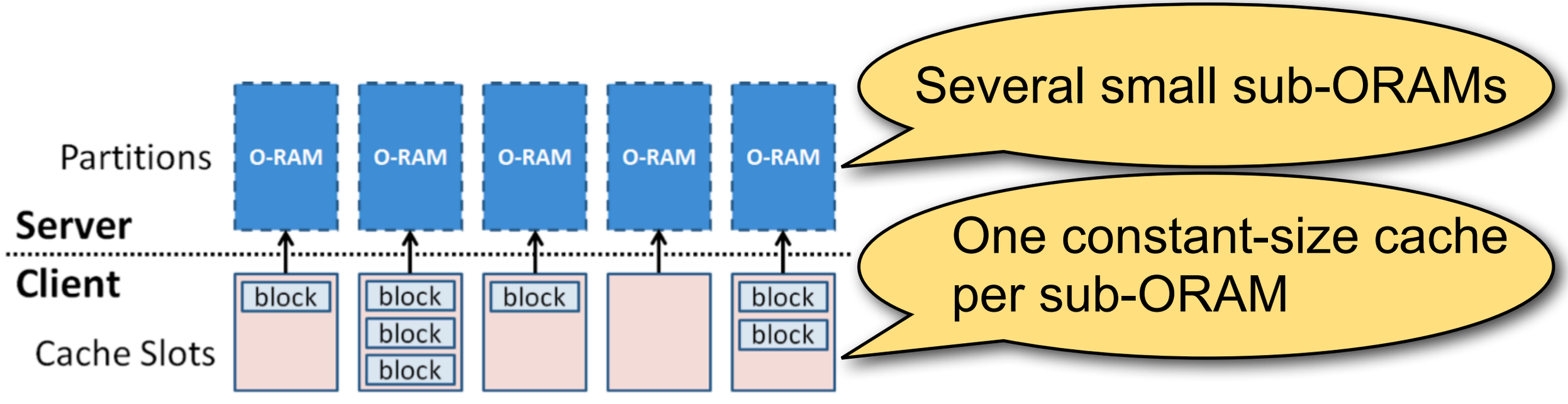
Further Improvements

- Can be de-amortized (GMOT'11a)
- Can use single O(log N)-size stash: (GM'10)
- Slightly faster shuffling: $O(\log^2 N / \log \log N)$ amortized (KLO'10)
- O(log N) overhead with client memory to store $O(N^{\delta})$ slots (GMOT'11b)

Outline

1. Goldreich's "Square Root" ORAM & Extensions
2. Ostrovsky's "Hierarchical" ORAM & Extensions
3. Cuckoo Hashing ORAMs: Scheme and Attack
4. **A "Practical ORAM"**

Practical ORAM: Data Layout [Stefanov, Shi, Song'12]



Client needs O(N) storage for position map!

... but it is storing log N bits per slot instead of the slot itself.

Read/Write Operation

Read/Write(addr)

- Look up slot index in position table
- Check local bucket at that index
- Query sub-ORAM on server at that index for slot
- Assign slot a new index and put it new local bucket

Background process:

- Scan local cache buckets sequentially, writing contents to corresponding sub-ORAM on server.
- As needed, shuffle sub-ORAMs locally using shuffle buffer.

Practical ORAM Performance

Configuration: $N^{1/2}$ sub-ORAMs implemented with modified GM'10, each of capacity about $N^{1/2}$.

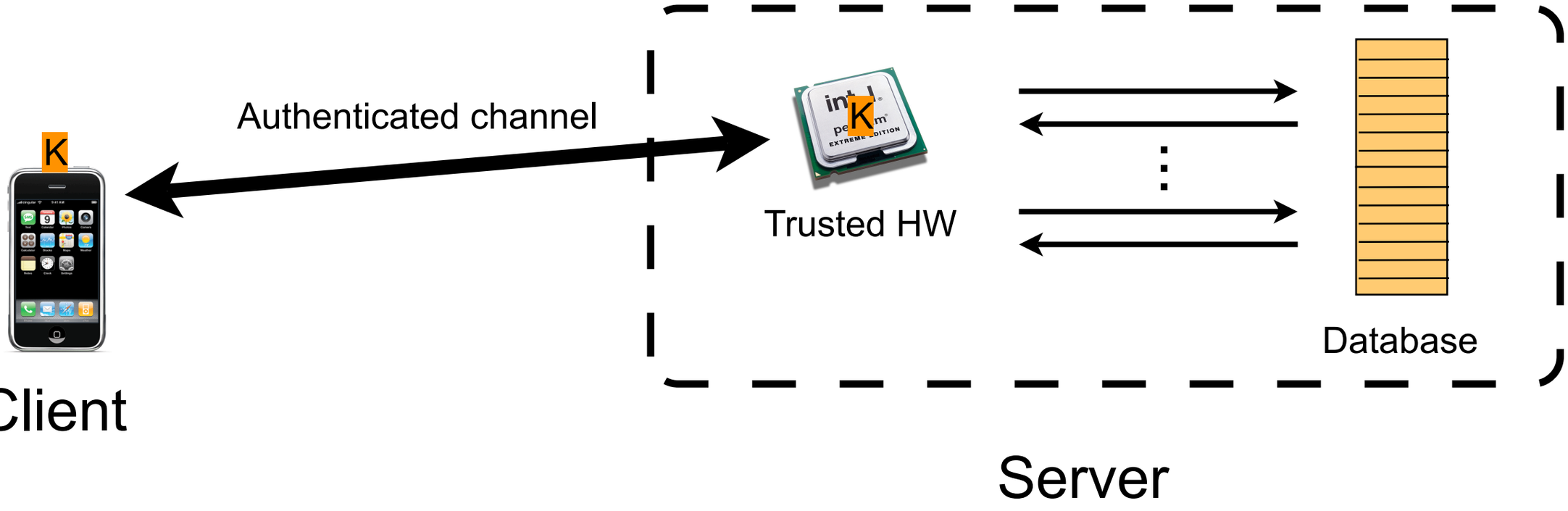
O-RAM Capacity	# Blocks	Block Size	Client Storage	Server Storage	$\frac{\text{Client Storage}}{\text{O-RAM Capacity}}$	Practical Performance
64 GB	2^{20}	64 KB	204 MB	205 GB	0.297%	22.5X
256 GB	2^{22}	64 KB	415 MB	819 GB	0.151%	24.1X
1 TB	2^{24}	64 KB	858 MB	3.2 TB	0.078%	25.9X
16 TB	2^{28}	64 KB	4.2 GB	51 TB	0.024%	29.5X
256 TB	2^{32}	64 KB	31 GB	819 TB	0.011%	32.7X
1024 TB	2^{34}	64 KB	101 GB	3072 TB	0.009%	34.4X

- Numbers are from round robin access of all blocks three times
- I'm not sure I understand security claims - proof appears to allow 1/poly advantage. Concrete numbers may be ok.
- Other schemes might be efficient after heavy optimization (?)

Outline

1. Goldreich's "Square Root" ORAM & Extensions
2. Ostrovsky's "Hierarchical" ORAM & Extensions
3. Cuckoo Hashing ORAMs: Scheme and Attack
4. A "Practical ORAM"
5. **Bonus: Hardware-assisted ORAM**

Hardware Assisted PIR



- Trusted HW acts as ORAM emulator for client
- Database acts of ORAM main memory

Implementations

- Asonov'04: Trivial ORAM
- Iliev-Smith'04: Square-root ORAM
- Wang-Ding-Deng-Bao'06: Square-root ORAM w/ different cache size
- Challenges here appear to be working with limited trusted hardware
- Crypto/theoretical contributions are secondary

Thoughts and Questions

- Prove the de-amortizing trick works as a black box for ORAMs of a certain form?
- Can (should) we simplify analysis via composition results?
  - **Parallel composition** without shared state (SSS'12 does this)
  - **Sequential composition**: one ORAM stores the state of the next ORAM (SCSL'11 does this)
- A more detailed practical analysis seems necessary. Taking slot size into account is important.
- To actually implement this stuff securely, you'd have to be **really** careful about timing attacks.
- Most of these ORAMs are variants on old ideas.
  - New approaches for optimal construction?

The End