Oblivious Computation with Data Locality

Gilad Asharov — CORNELL TECH

Hubert Chan — 香 港 大 學 THE UNIVERSITY OF HONG KONG

Kartik Nayak — UNIVERSITY OF MARYLAND

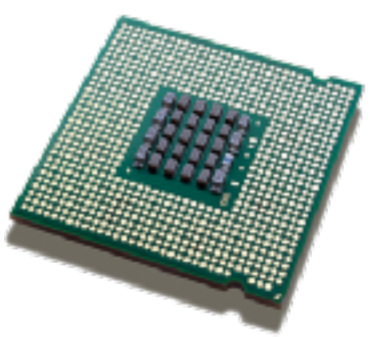
Rafael Pass — CORNELL TECH

Ling Ren — Massachusetts Institute of Technology

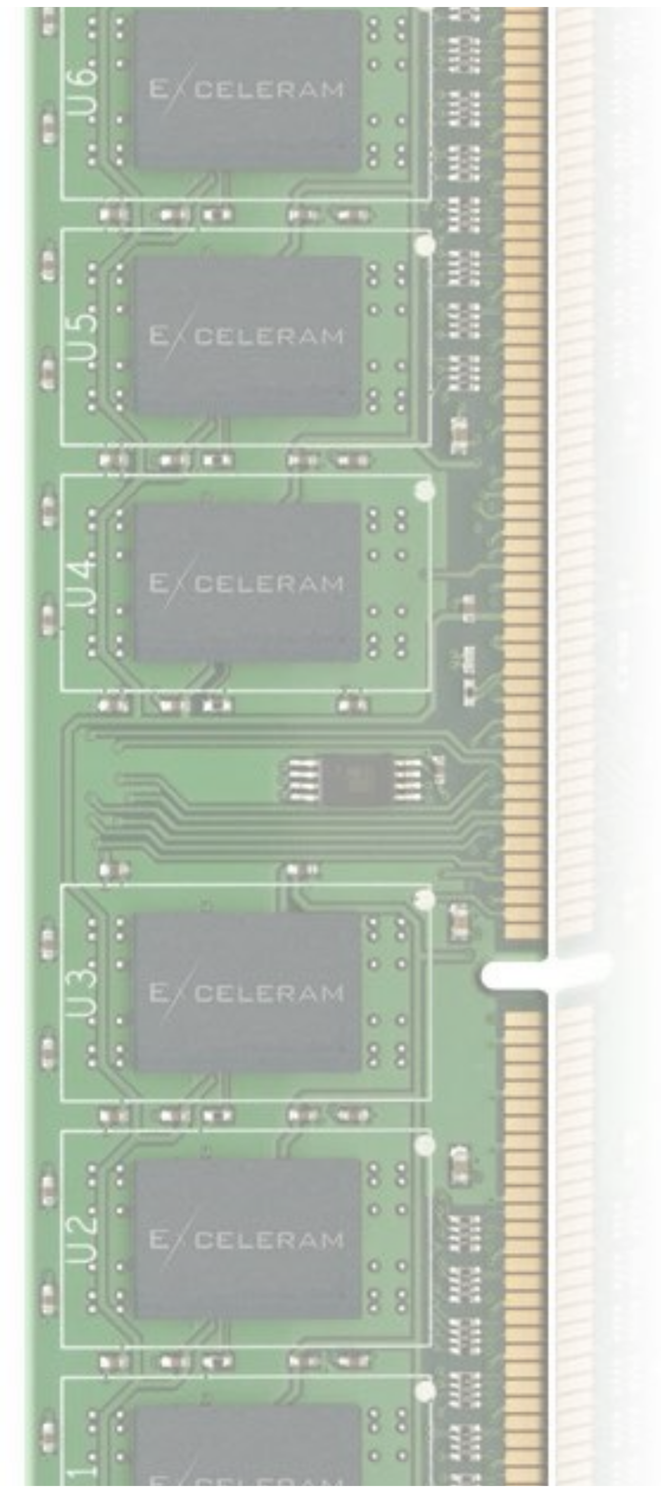
Elaine Shi — Cornell University

Access Pattern Leakage

(or, why encrypting the data is insufficient?)

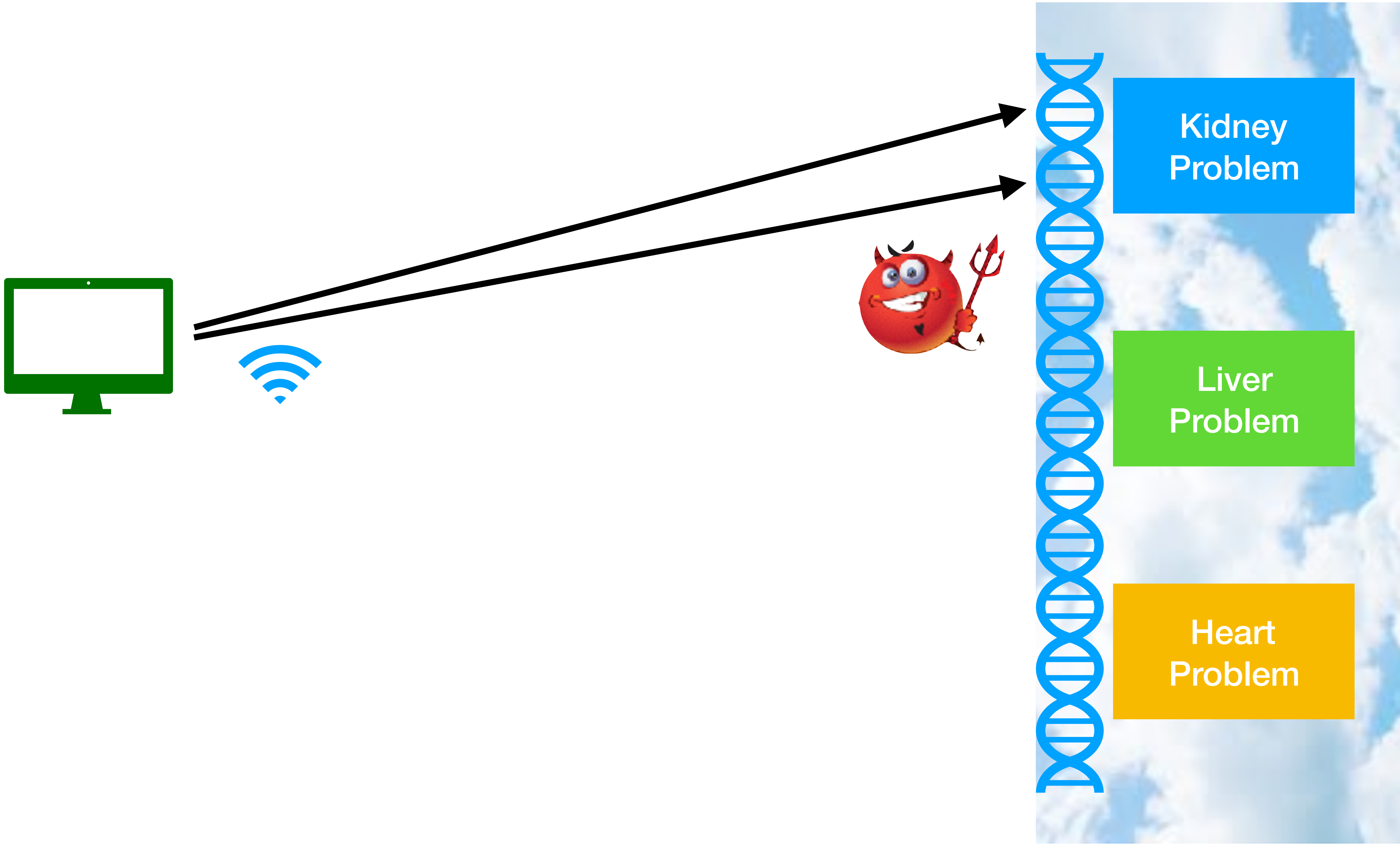


secure processor



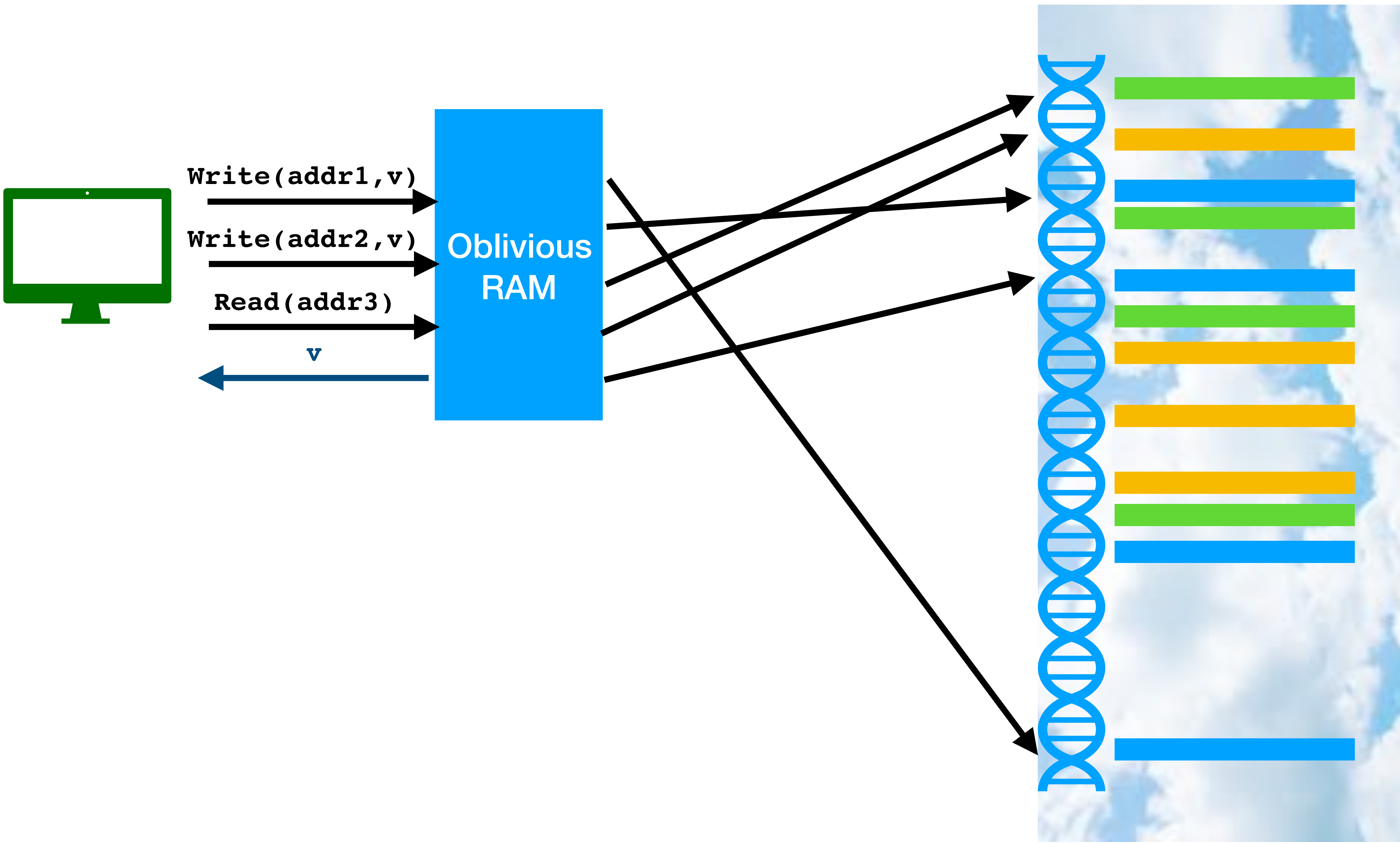
Access Pattern Leakage

(or, why encrypting the data is insufficient?)



Oblivious RAM

(or - How to Hide the Access Pattern?)

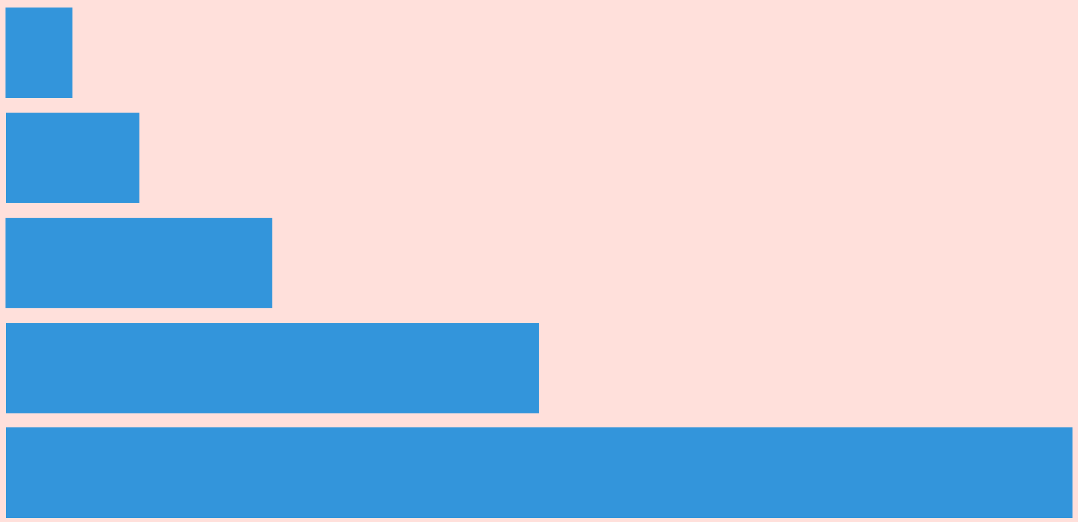


Oblivious RAM

- Introduced by Goldreich [STOC’87]
- **Informal definition:**
  - The access pattern can be simulated by the total number of **`Read/Write`** instructions that the program performs

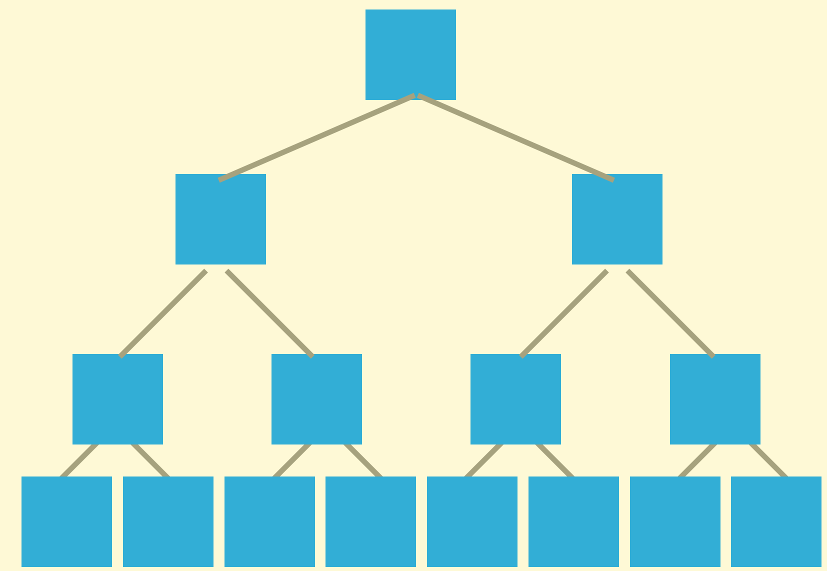
- **Lower bound:** memory N
  - $\Omega(\log N)$ overhead for every operation
    - Recently - very interesting progress [GO96,BN16,LN18]

Known ORAMs



Hierarchical

[GO96,Kushilevitz,Lu,Ostrovsky12]



Tree based ORAM

[Stefanov,van Dijk,Shi,Chan,
Fletcher,Ren,Yu,Devadas13]

$$\sim O(\log^2 N)$$

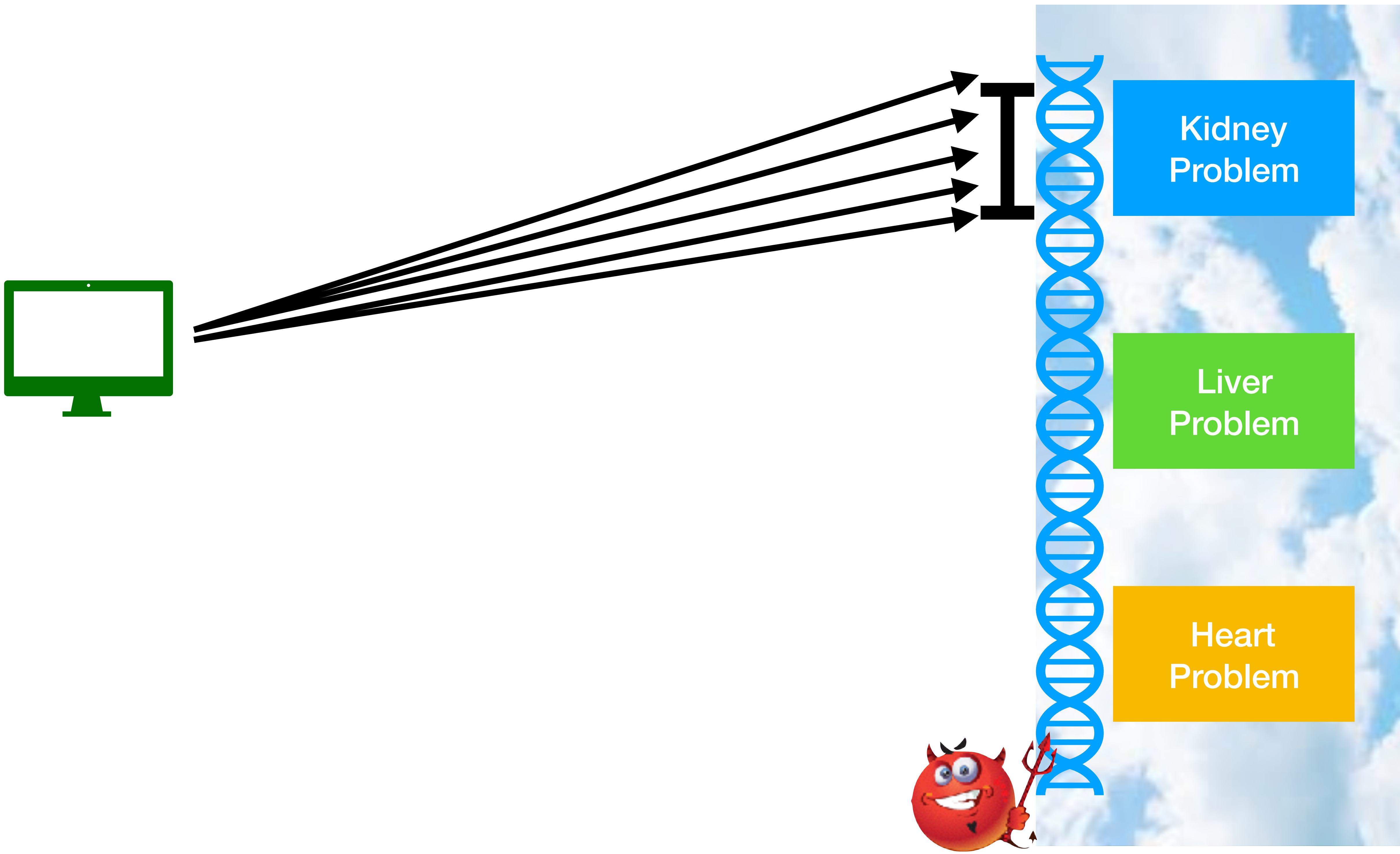
Locality

- **A phenomenon:**

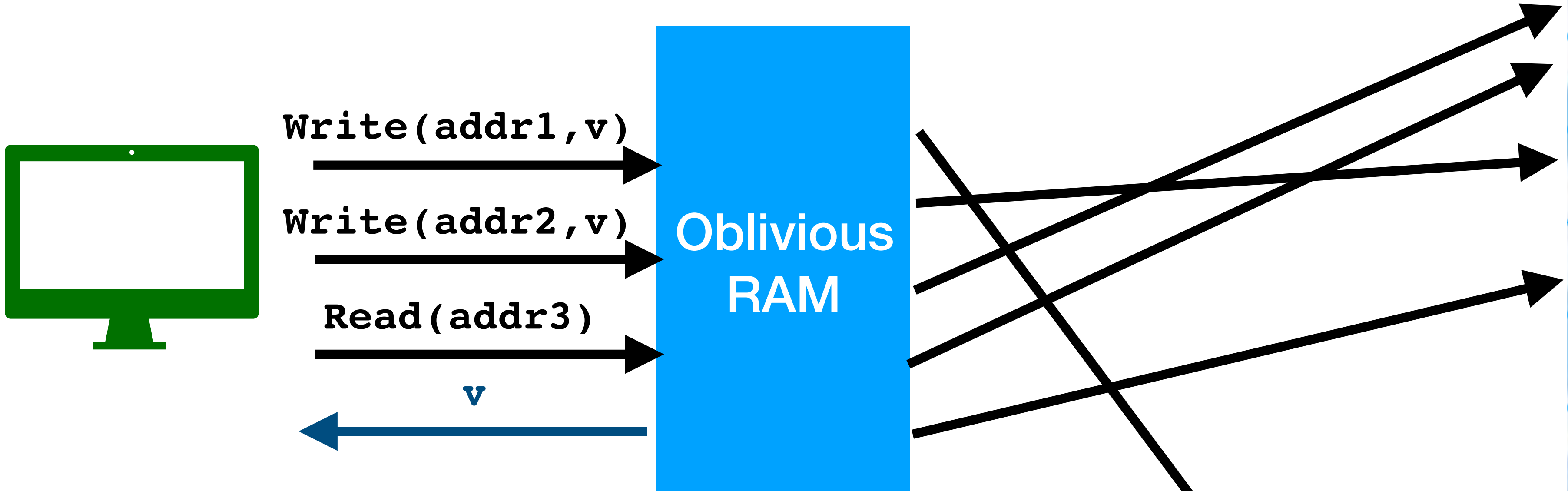
if a program or application accesses some address it is very likely to access also a neighboring address

- **Locality is everywhere:**
  - **Physically**: Rotational hard-drive are significantly faster when accessing sequential data than random seeks
  - **Cache**: Usually fetching neighboring data as well
    - Surfaced from implementations of Searchable Symmetric Encryption
- **A crucial efficiency measure!**

Accessing Sequential Data?



Accessing Sequential Data?



- **ORAM completely destroys the locality of the program!**
- Accessing a **single** contiguous-region of size **L** results in
  accessing **O(L log²N)** non-contiguous blocks

Our Goal:
ORAM with Locality

- ORAM that preserves the locality of the program:
  - If an incoming request access a possibly ***large contiguous*** region, then the ORAM should also access ***contiguous memory*** regions

- **Locality** and **obliviousness** are contradicting goals!
  - ORAM **must** shuffle the data around the memory
  - Locality is usually achieved by **highly structured memory layout**

Related Work

- Locality in algorithms […Vitter01]
- SSE does not scale well to big databases without considering **locality** [CJJKRS,CRYPTO'13]
  - Tradeoffs between obliviousness, space and locality
  - [Cash,Tessaro'14],[**A**,Naor,Segev,Shahaf'16],[Demertzis,Papamanthou'17],[**A**,Segev,Shahaf'18],[Demertzis,Papadopoulos,Papamanthou'18]
- Oblivious RAM and secure computation
  - [Gordon,Katz,Kolesnikov,Krell,Malkin,Raykova,Vahlis'12],[Gentry,Goldman,Halevi,Lu,Ostrovsky,Raykova,Wichs'14],[Wang,Huang,Chan,shelat,Shi'14]
  - Garbled RAM [LuOstrovsky13,…]
  - Avishay's talk (next)

Agenda

- Defining locality
- **Impossibility result**
- **Primitive I:** Range ORAM
- **Primitive II:** File ORAM
- Locality-friendly oblivious sort

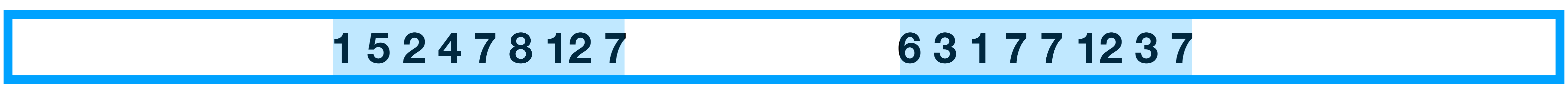
Defining Locality

- **Locality:** intuitively, number of sequential **memory regions** accessed during the execution of the program

Locality = 3

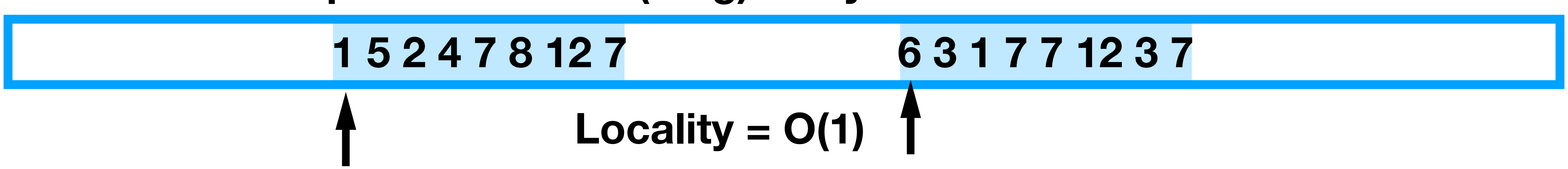
1 disk, minimize “move” of the read/write head

Inner product of two (long, say n) arrays?



Locality = O(n)

Inner product of two (long) arrays — 2 read/write heads?



Locality = O(1)

Defining Locality

- We allow accessing **H** regions concurrently
  - Think of **H** different disks, or
  - A cache with **H** different lines, or
  - A disk with **H** read/write heads

Definition:
An algorithm / program is **(H,L)-local**
if it performs **L** sequential read/writes from a memory that is equipped with **H**-heads

- **Good locality** = small **H** (O(1)), small **L**

Impossibility Result*

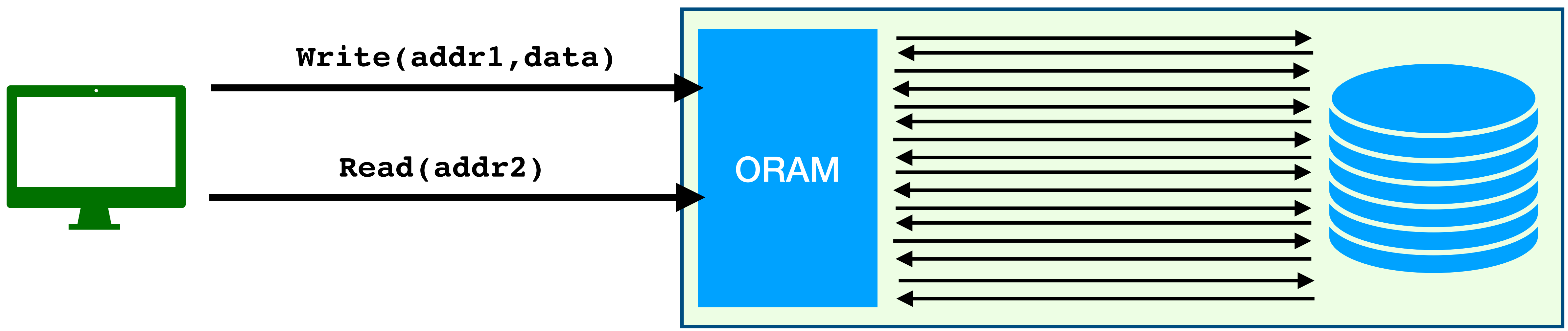
- Local ORAM is **impossible**
  - ORAM **must** randomly permute elements around the memory
  - Must hide whether we have
    L requests of *non-contiguous* blocks or
    a single request of **L** *contiguous* blocks

Theorem:
Any (O(polylogN),O(polylogN))-local ORAM scheme would have inefficient *bandwidth* blowup $\boldsymbol{\Omega(N^{1-\epsilon})}$ for some constant ϵ

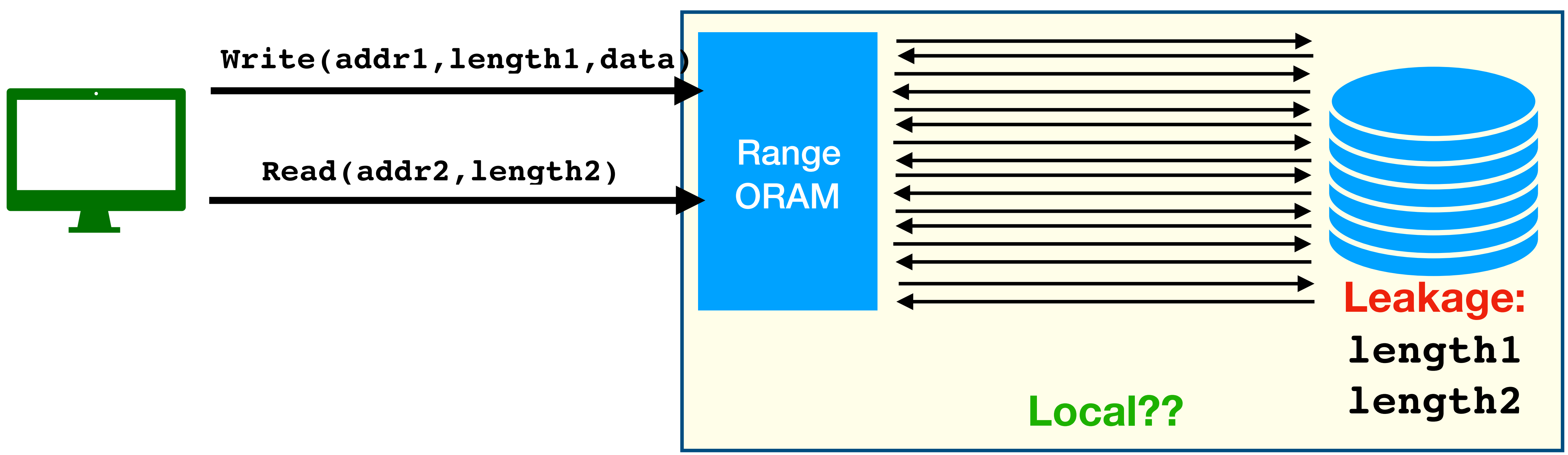
- We must relax our requirements
  - aka, leakage…

*In the balls and bins model

First Primitive: Range ORAM



Simulator receives number of read/write operations



Simulator receives length1,length2...

Our Results

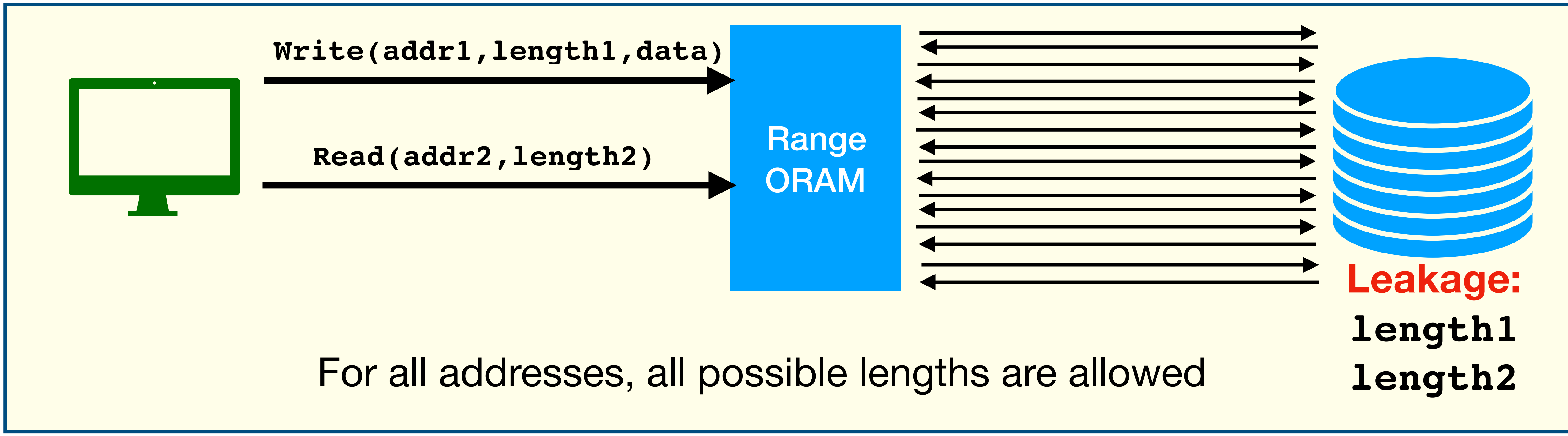
- **Impossibility**: locality without leakage of lengths

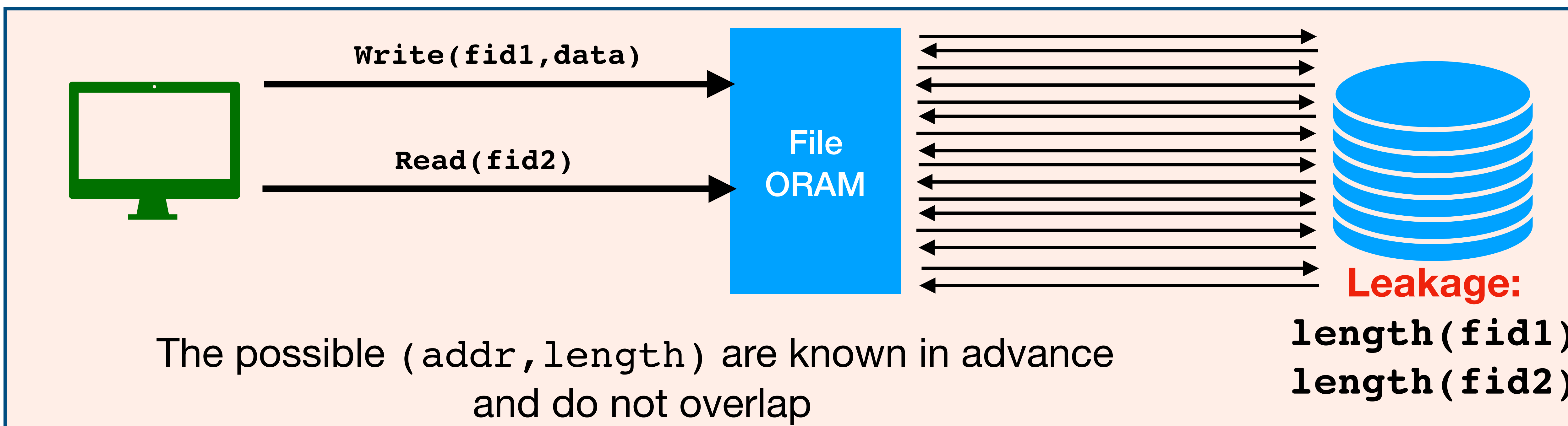
	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	**stat**	**O(NlogN)**	**L $\tilde{O}(\log^3 N)$**	**$\tilde{O}(\log^3 N)$**	**L**
ORAM	stat	O(N)	L $o(\log^2 N)$	L $o(\log^2 N)$	none

On Leaking the Lengths

- **Inherent**: our lower bound…
- Strict generalization of ORAM
  - The client can choose *when* and *what* to leak
- In many applications, *ordinary ORAM also leaks sizes* when accessing a region of length L
  - via communication volume [KellarisKolliosNissim16]
- **Possible extension**:
  add differential privacy to mitigate the leakage

Second Primitive: File ORAM





Our Results

- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	stat	$O(N\log N)$	$L\,\tilde{O}(\log^3 N)$	$\tilde{O}(\log^3 N)$	L
File ORAM	**comp**	**$O(N)$**	**$L\,\tilde{O}(\log^2 N)$**	**$\tilde{O}(\log N)$**	**L**
ORAM	stat	$O(N)$	$L\,o(\log^2 N)$	$L\,o(\log^2 N)$	none

Essentially, locality for free!

Our Results

- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	stat	$O(N\log N)$	$L\ \tilde{O}(\log^3 N)$	$\tilde{O}(\log^3 N)$	L
File ORAM	comp	$O(N)$	$L\ \tilde{O}(\log^2 N)$	$\tilde{O}(\log N)$	L
ORAM	stat	$O(N)$	$L\ o(\log^2 N)$	$L\ o(\log^2 N)$	none

- An intermediate result: ***Locality-Friendly oblivious sort***
  - Perfect: $O(N \log^2 N)$-work and $(2, O(\log^2 N))$-locality
  - **Statistical: $\tilde{O}(N \log N)$**-work and **$(3, \tilde{O}(\log N))$**-locality

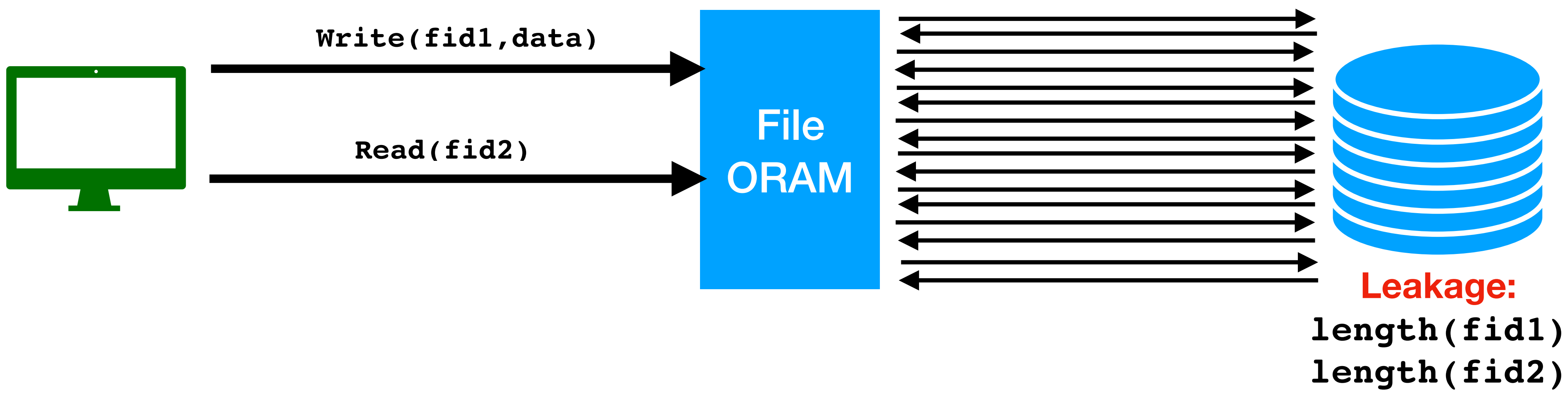
This Talk

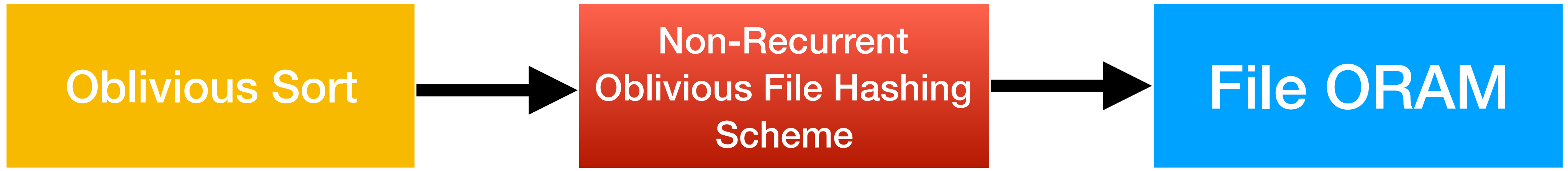
- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	stat	O(NlogN)	L Õ($\log^3$N)	Õ($\log^3$N)	L
File ORAM	**comp**	**O(N)**	**L Õ($\log^2$N)**	**Õ(logN)**	**L**
ORAM	stat	O(N)	L o($\log^2$N)	L o($\log^2$N)	none

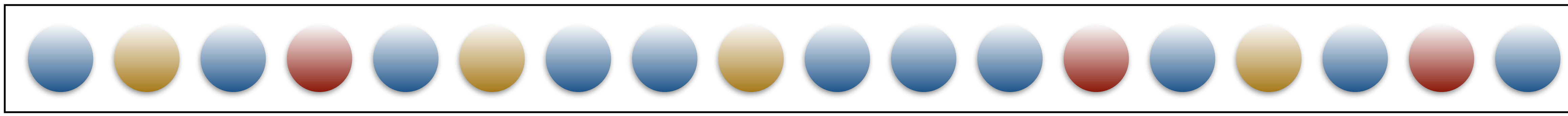
- An intermediate result: ***Locality-Friendly oblivious sort***
  - **Perfect:** **O(N $\log^2$N)**-work and **O($\log^2$N)**-locality
  - **Statistical: Õ(N logN)**-work and **Õ(logN)**-locality

File ORAM: Construction





Non-Recurrent File Hashing Scheme with Locality

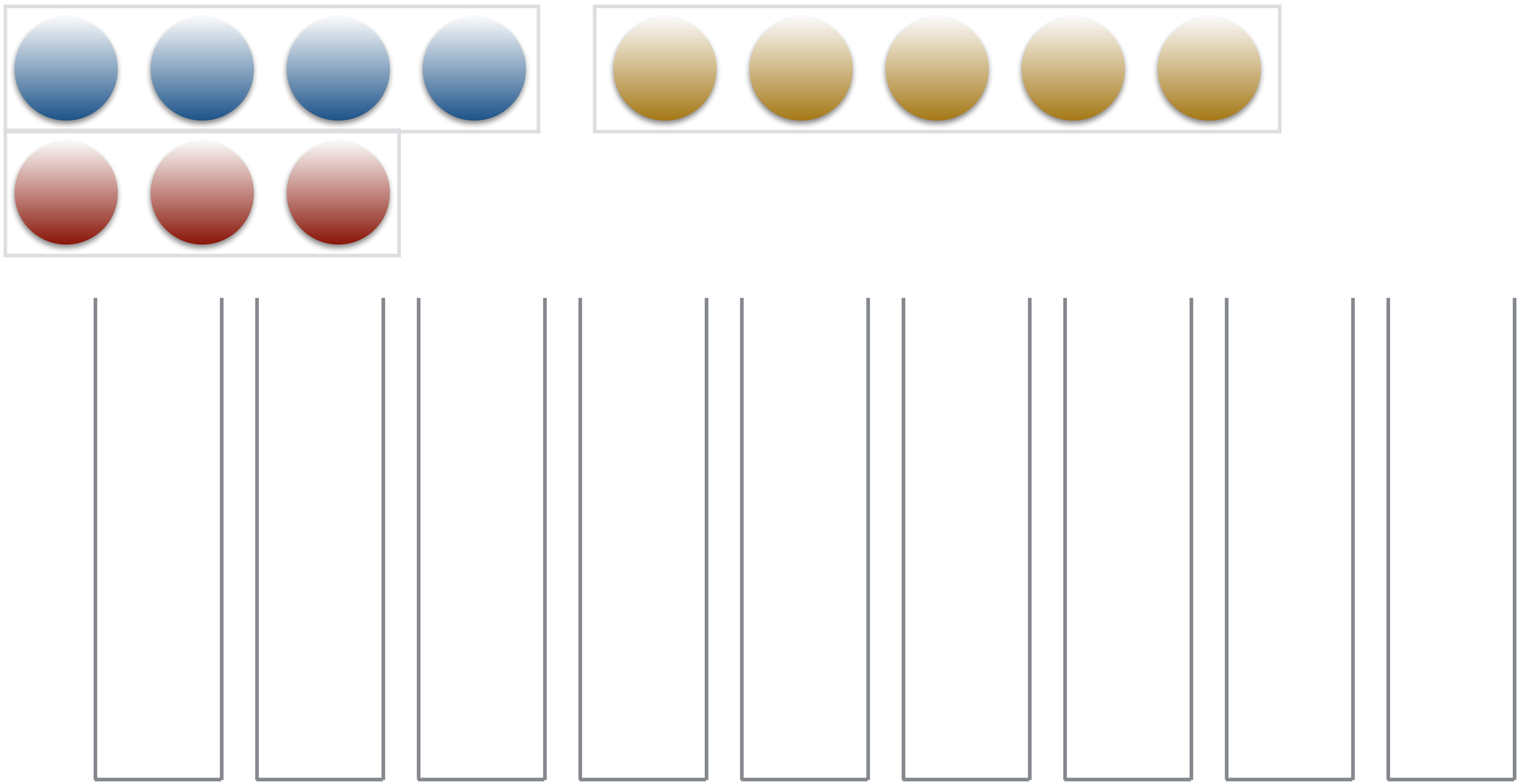


- **Functionality**:
  - **Build(X)** : Given an array with files data, build structure
    - Each element: (`fid,offset,data`)
  - **Read(fid,len):** returns all elements with `fid`
    Supports also fake `fid=*`

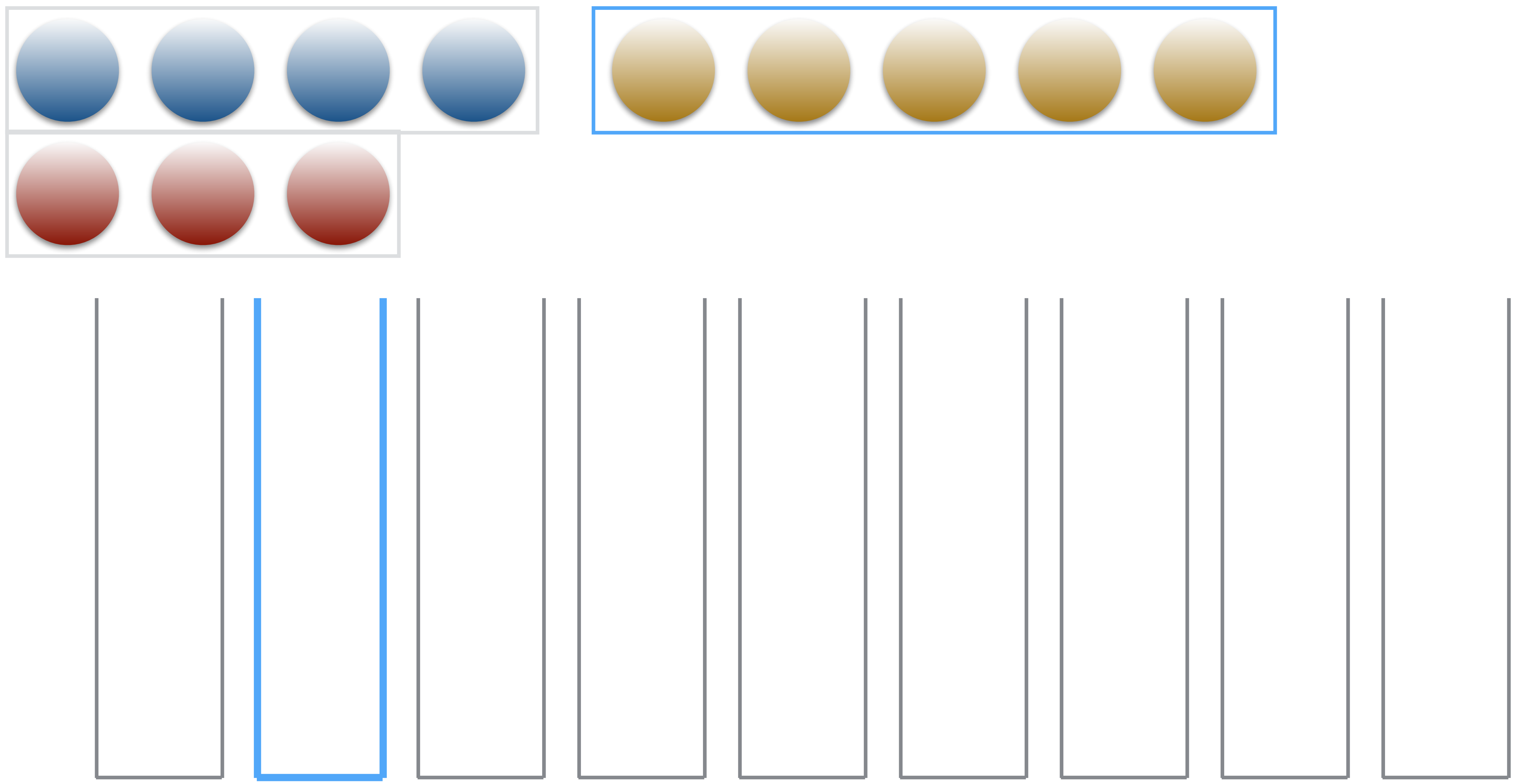
- **Obliviousness**: instructions
  `(Build(X),Read(fid1,len1),Read(fid2,len2),…,)`
  with non-recurrent `fid` (except for `*`) can be simulated from
  `(|X|,len1,len2,…)`

How to build such a primitive with “good” locality?

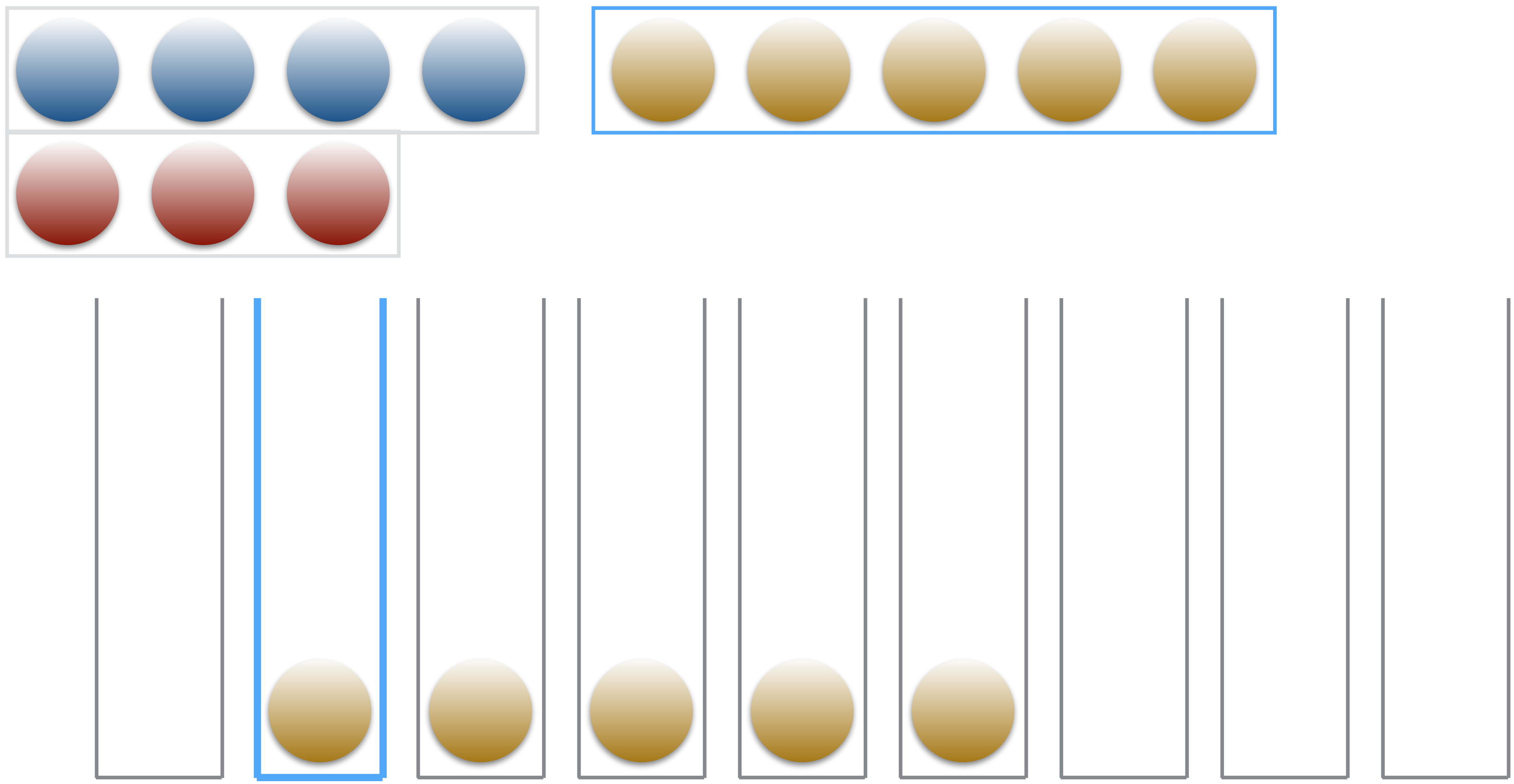
Two-Dimensional Allocation



Two-Dimensional Allocation

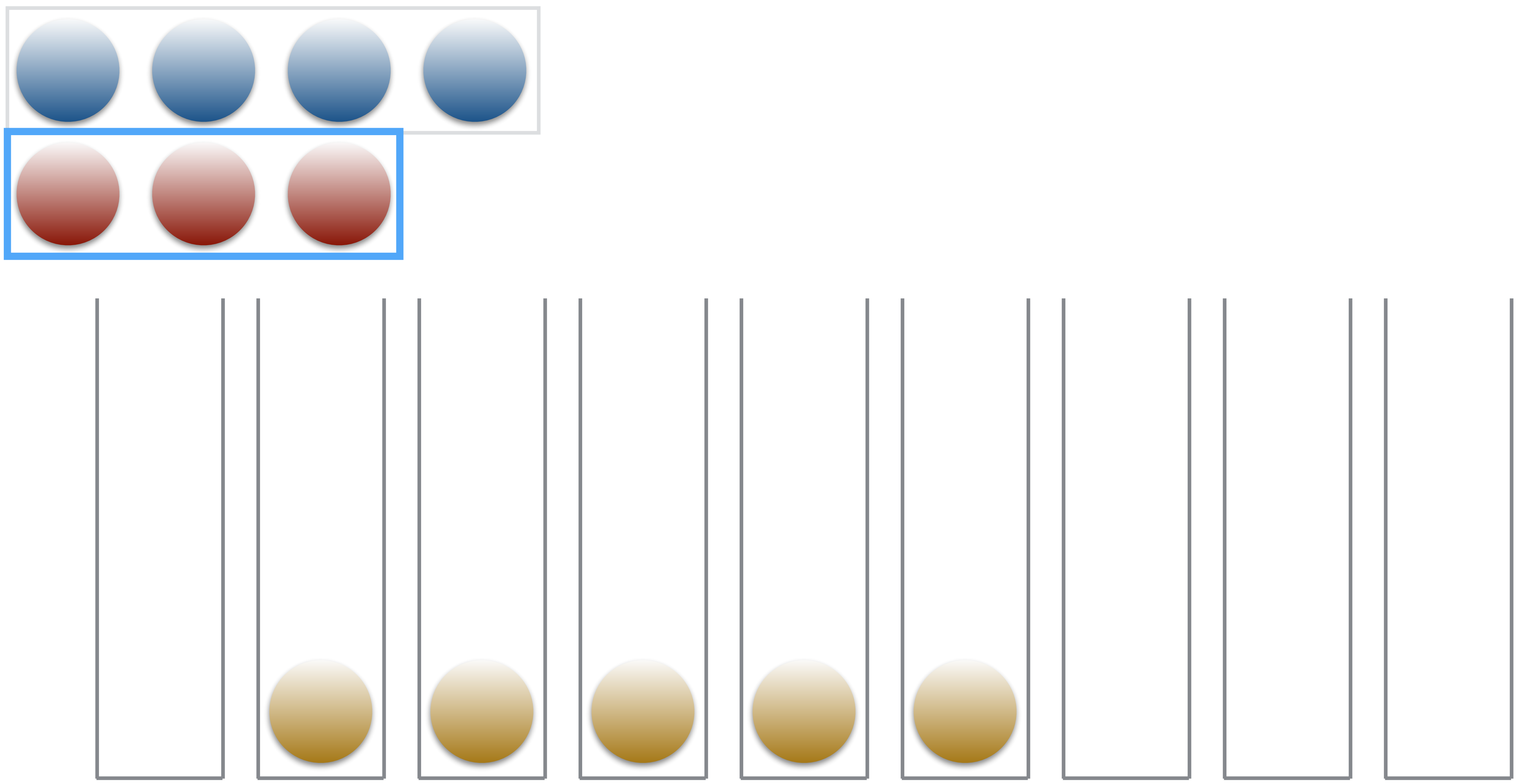


Two-Dimensional Allocation

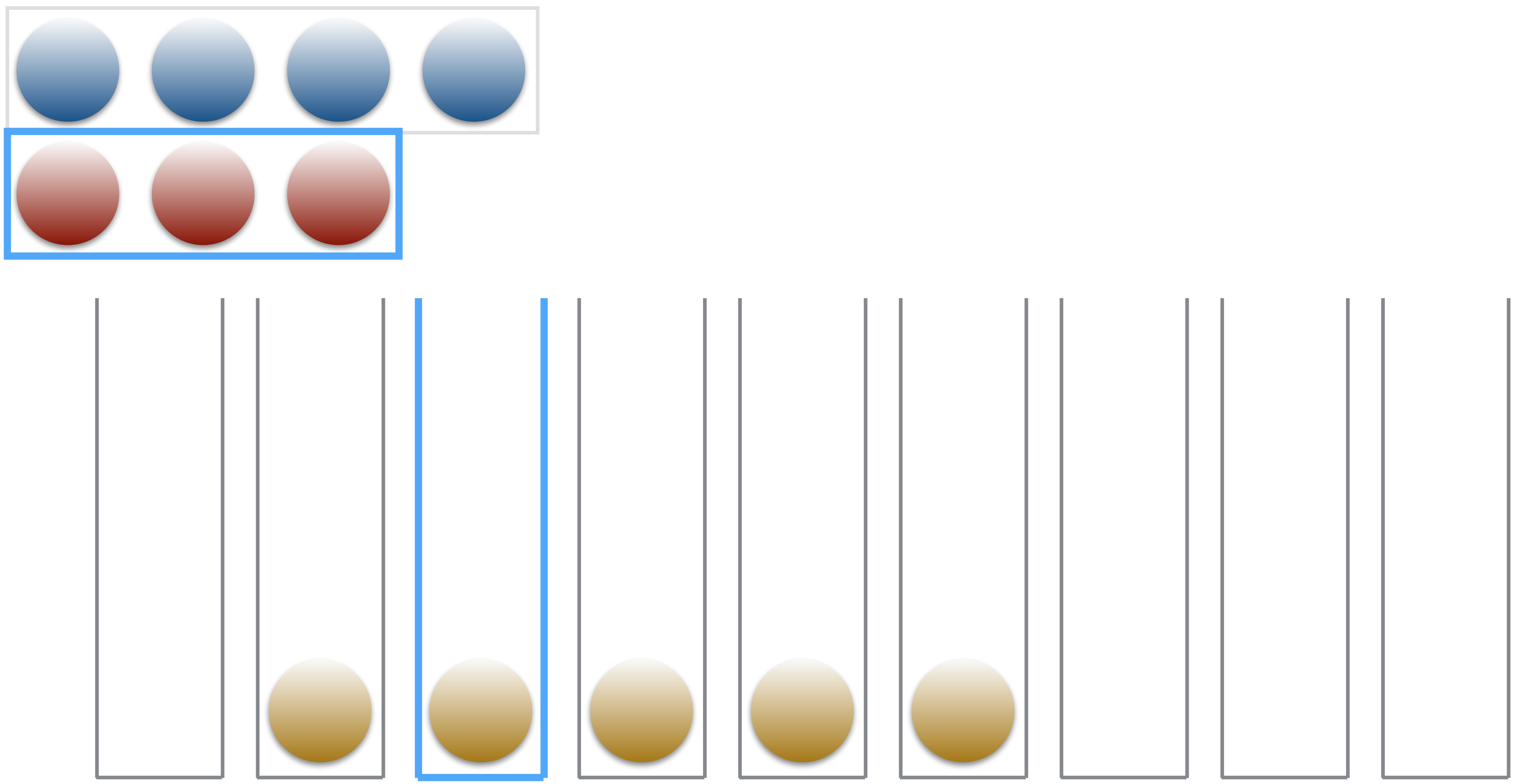


Place the whole file according to
a ***single*** probabilistic choice!

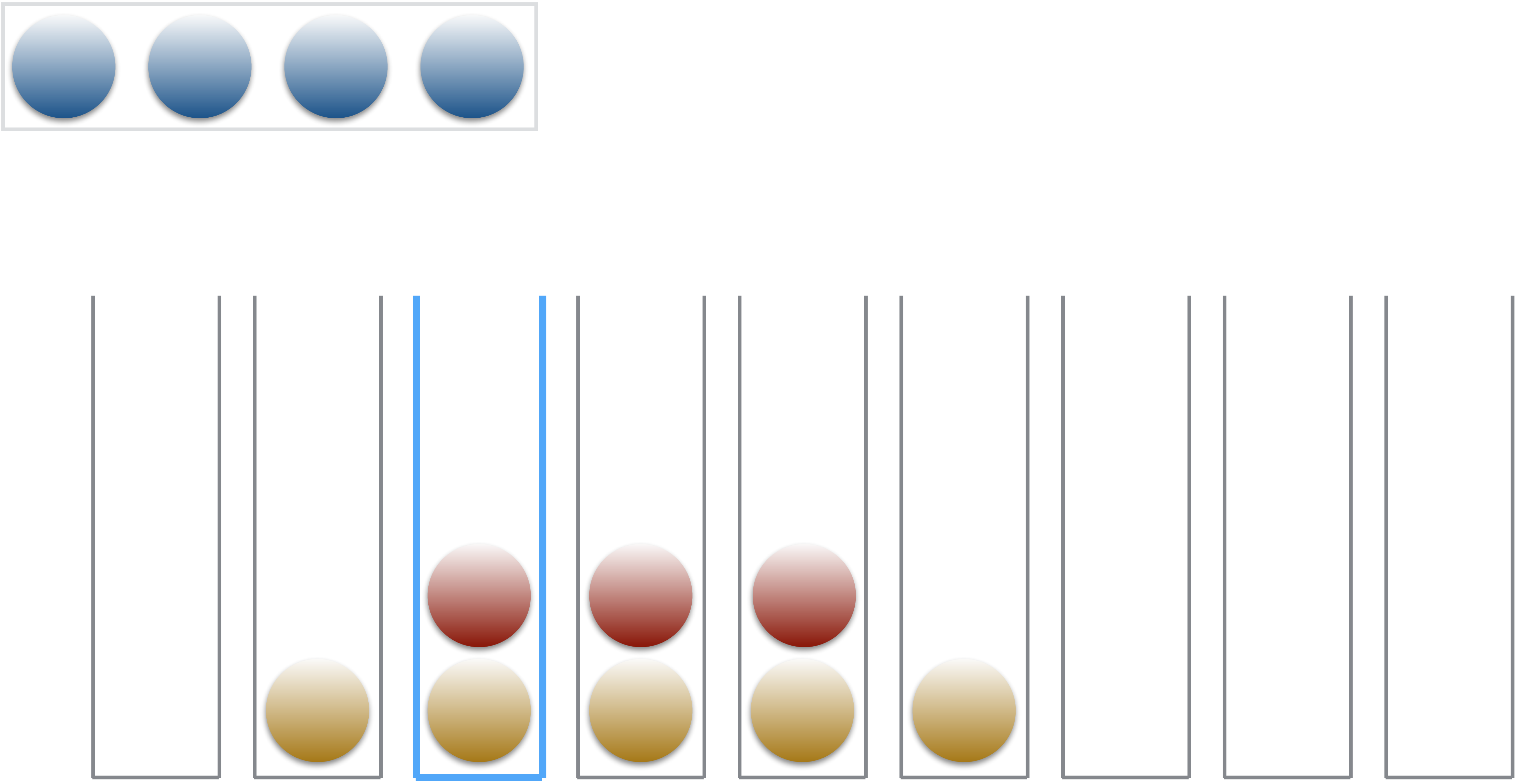
Two-Dimensional Allocation



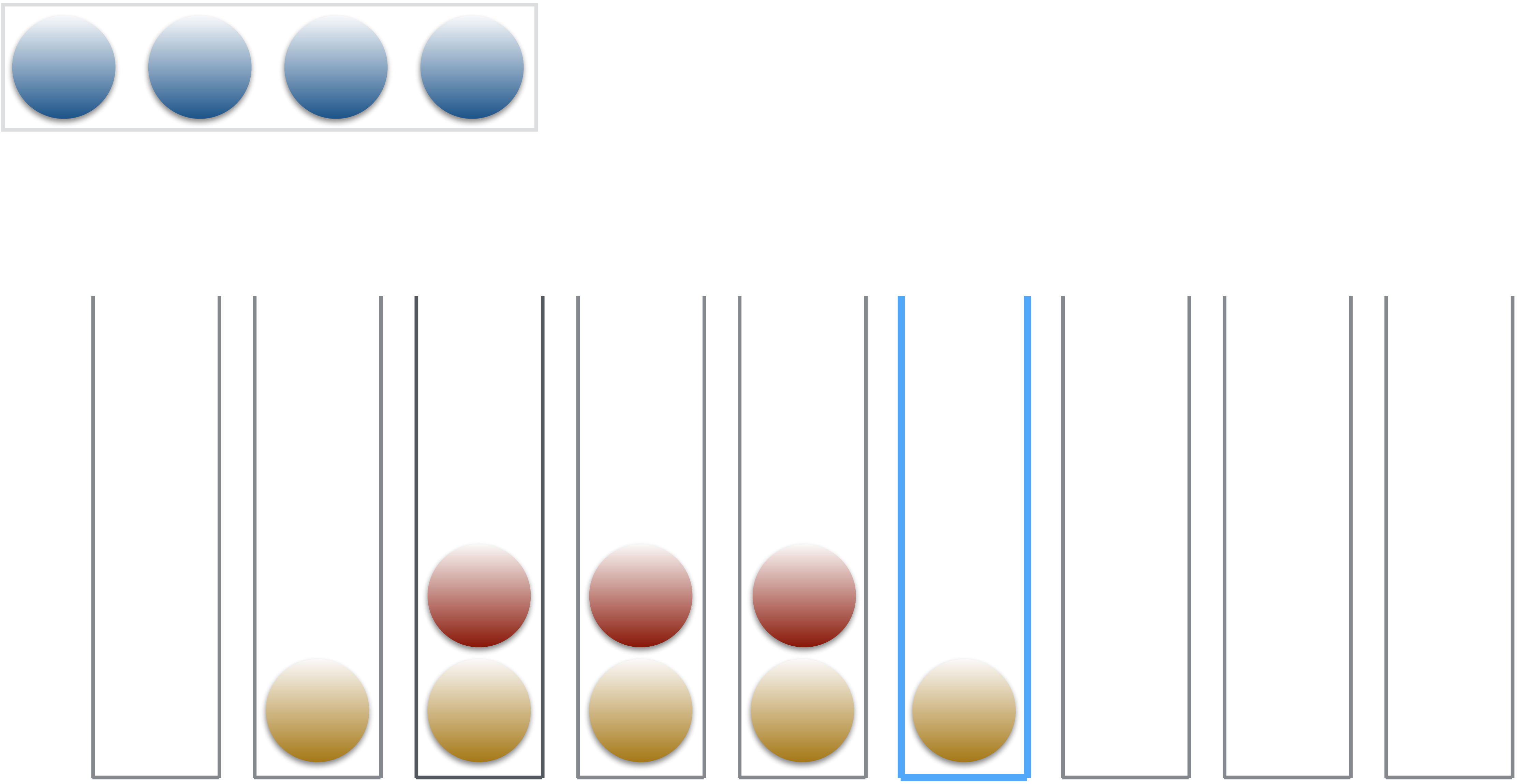
Two-Dimensional Allocation



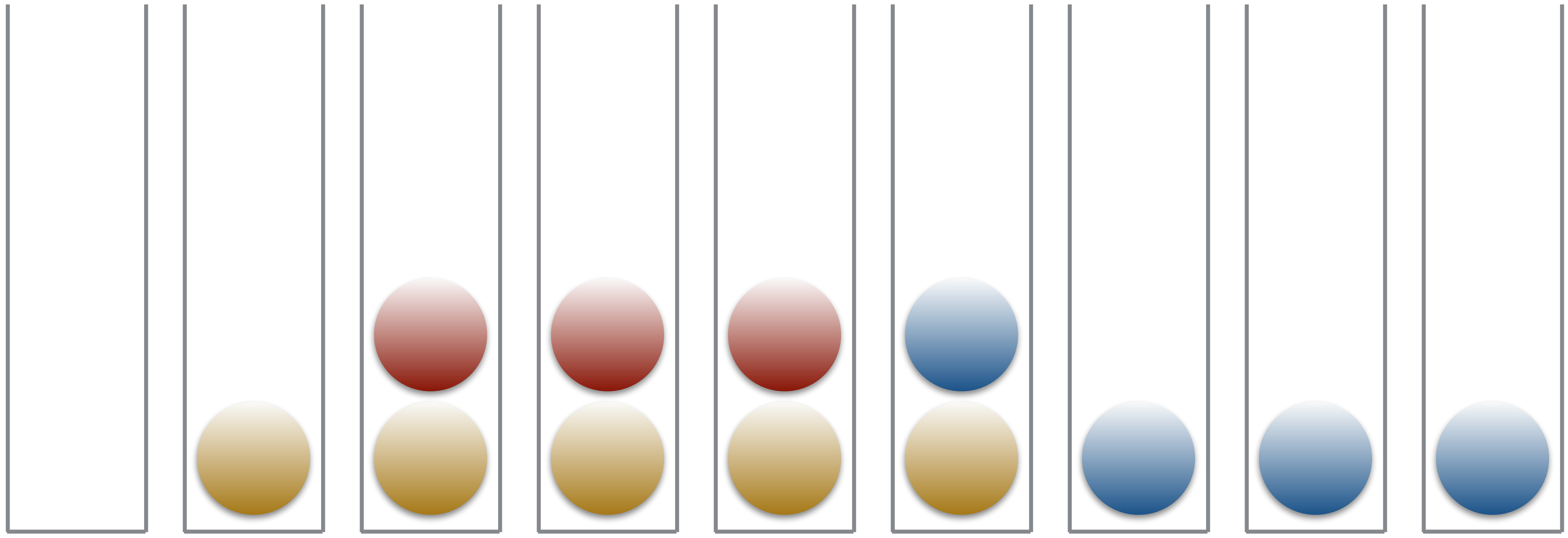
Two-Dimensional Allocation



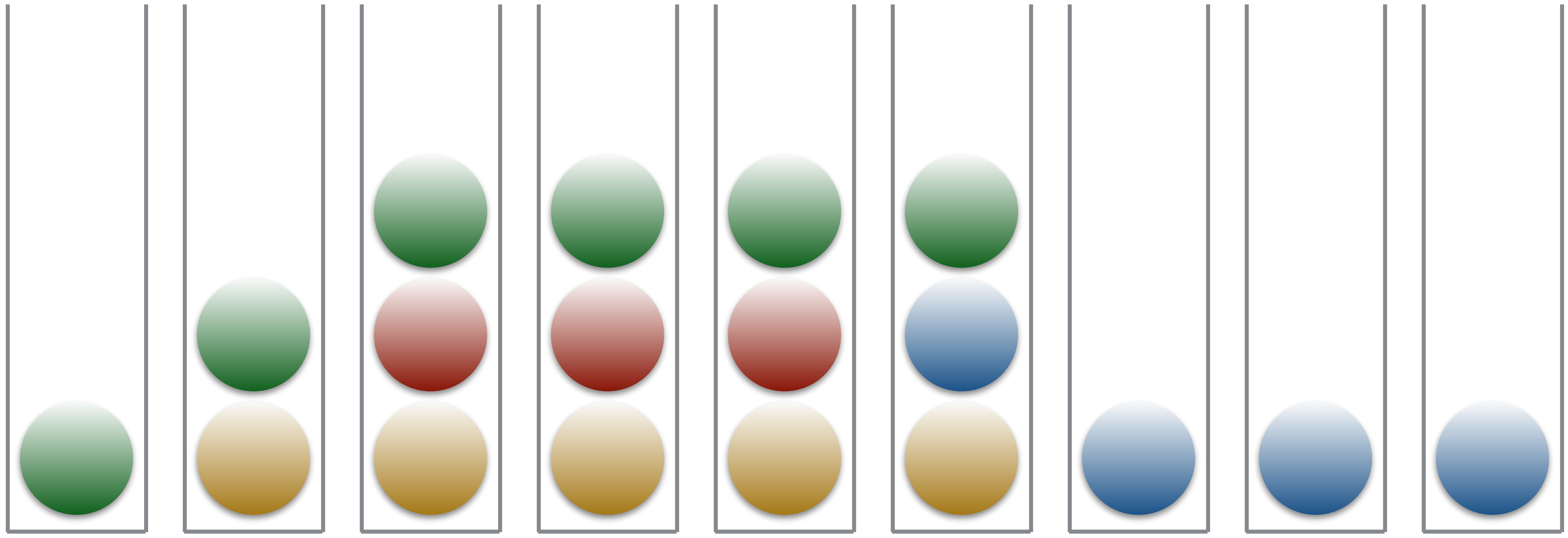
Two-Dimensional Allocation



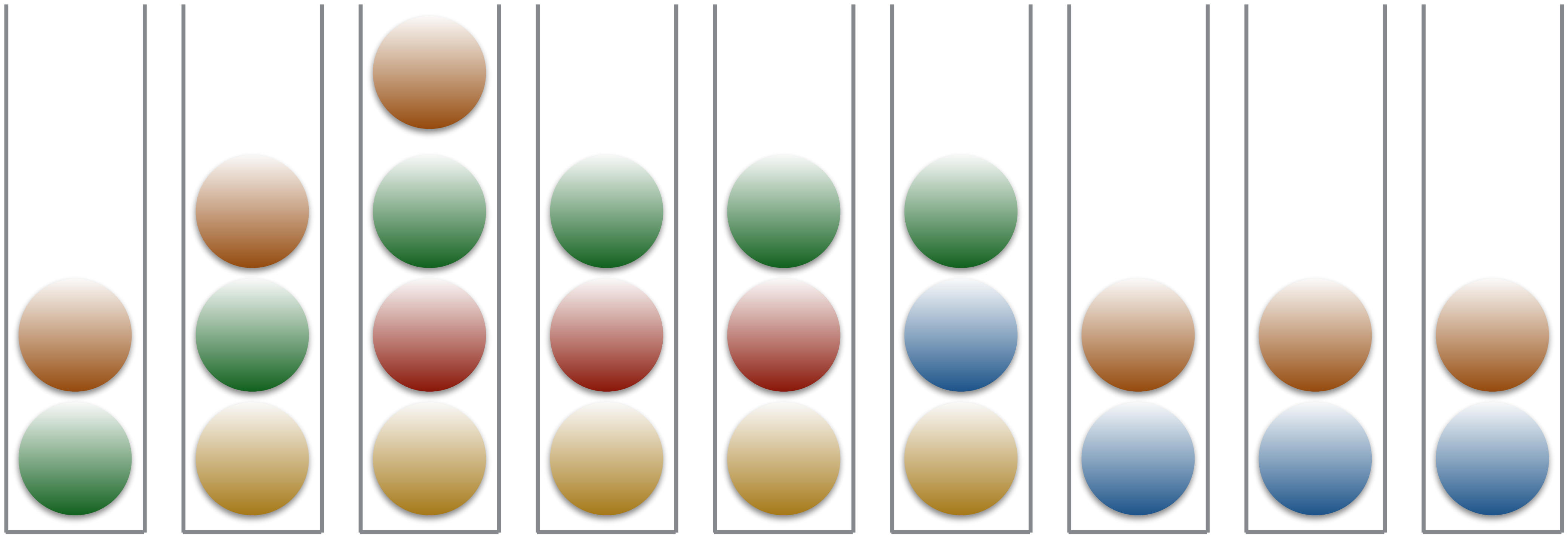
Two-Dimensional Allocation



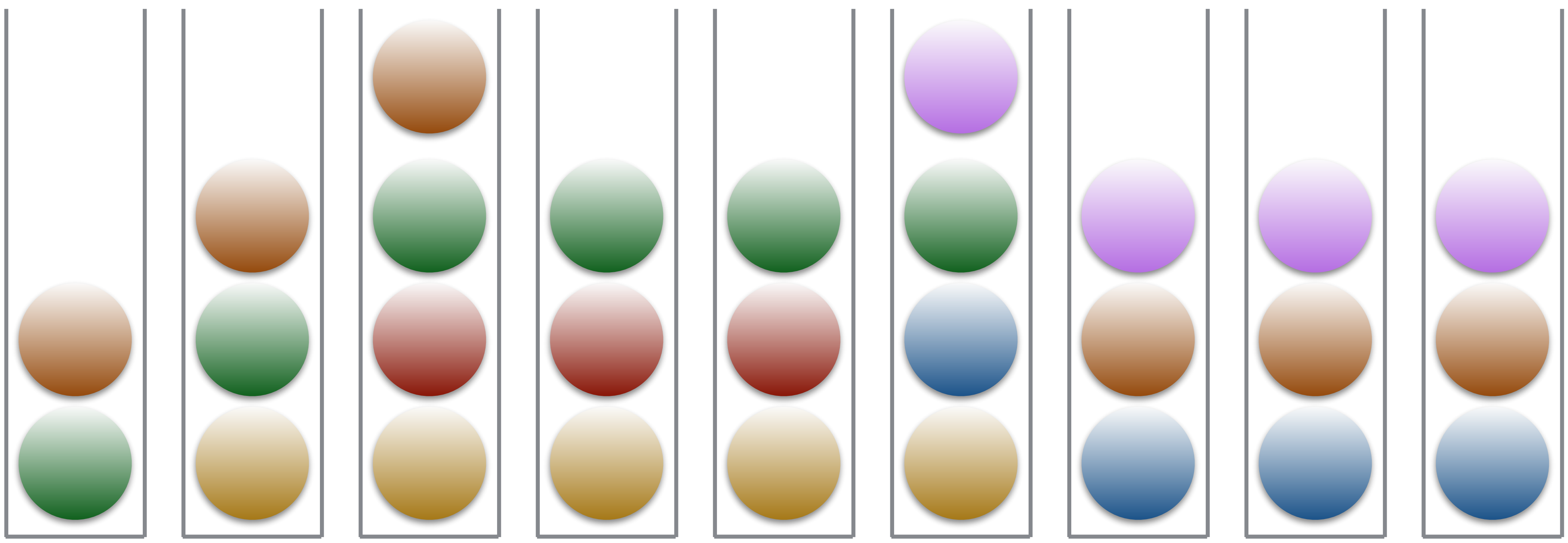
Two-Dimensional Allocation



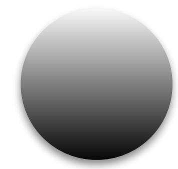
Two-Dimensional Allocation

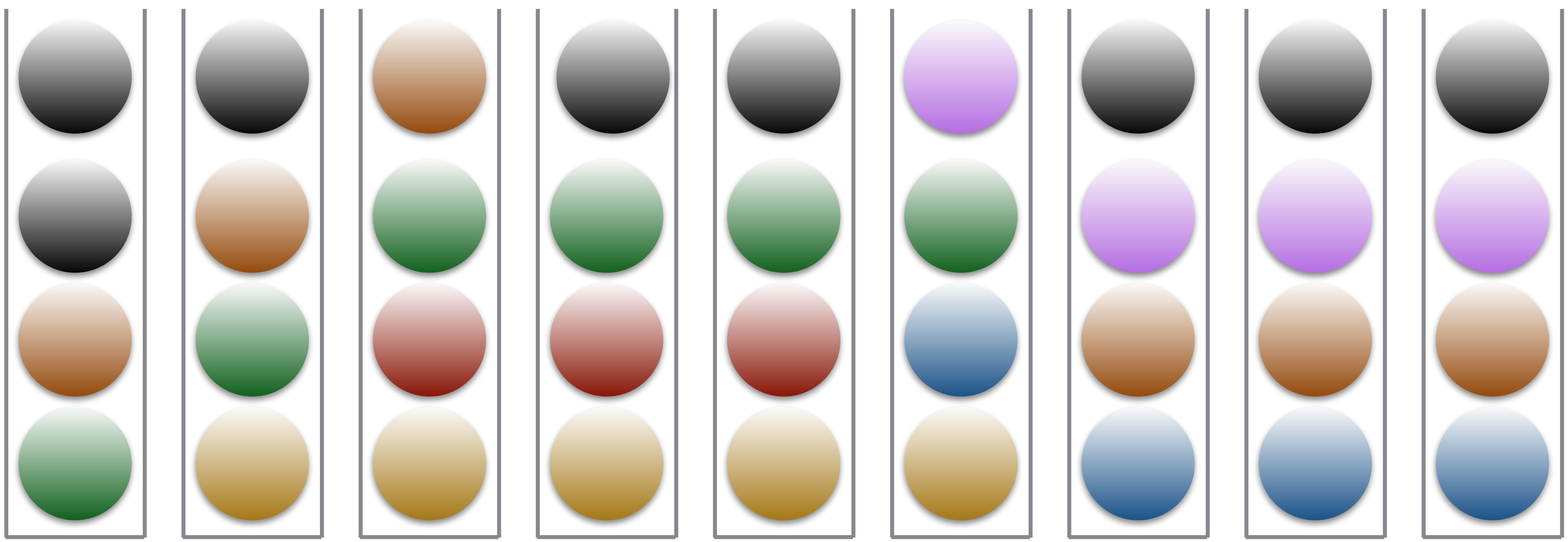


Two-Dimensional Allocation



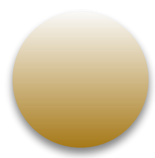
Two-Dimensional Allocation

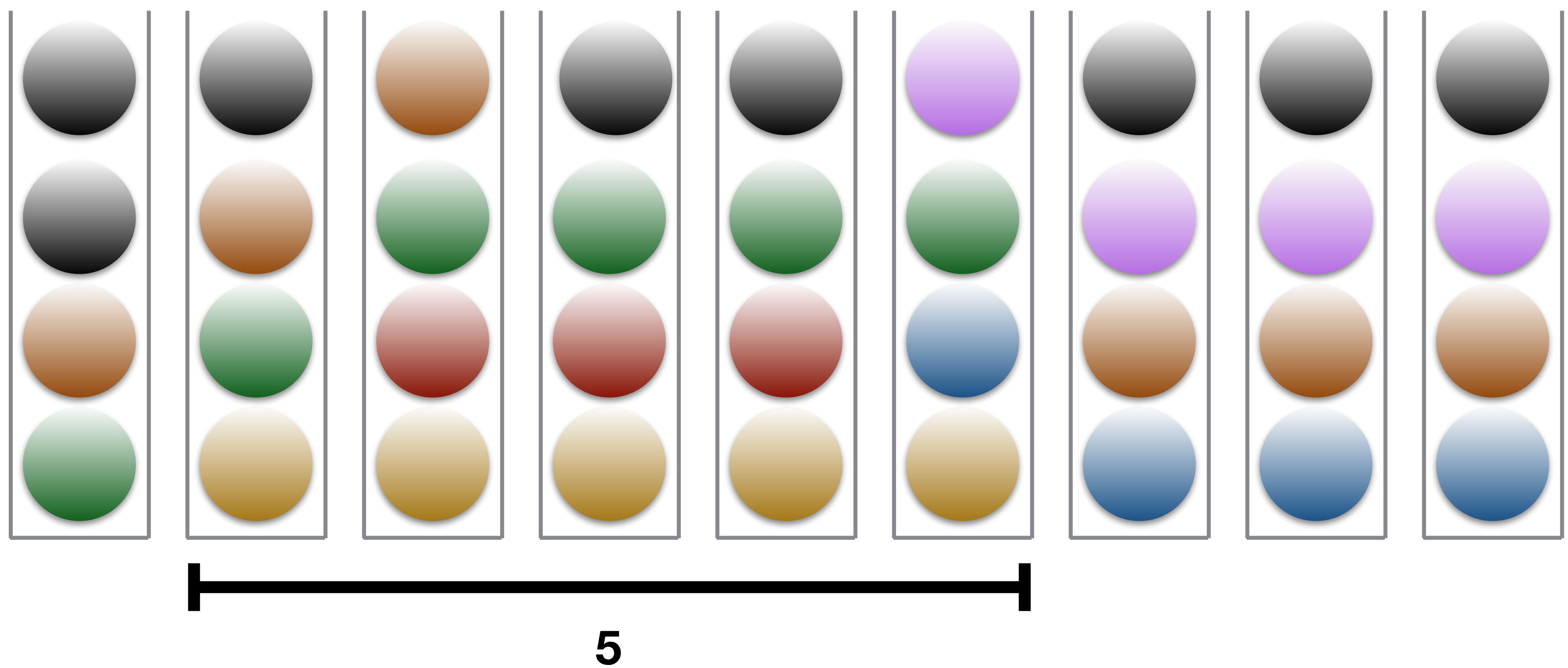
Pad with dummies 



What is the maximal load?

How Do We Search?

Read( ,5)

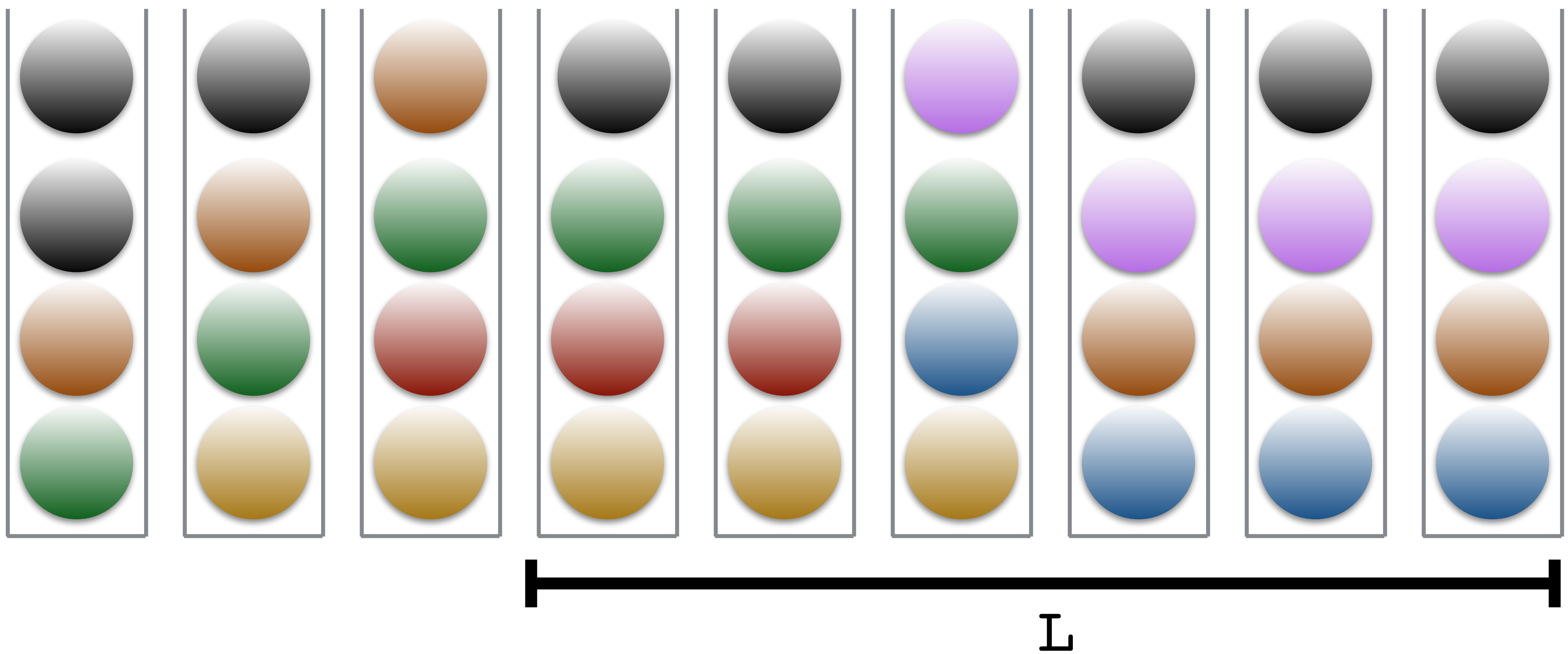


Overhead = bin size

How Do We Search?

Read(`*, L`)

(*=fake fid)



Just access random `L` consecutive bins

Overhead = bin size

Two-Dimensional Allocation

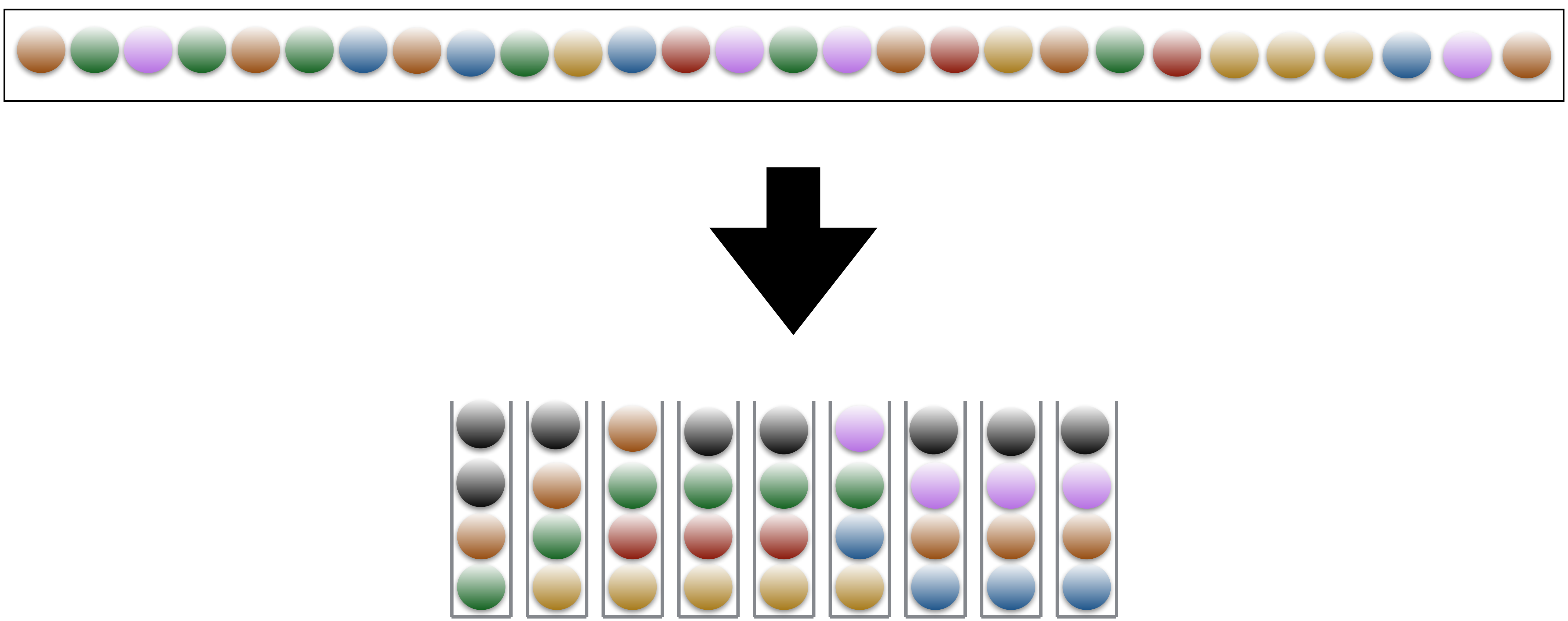
[AsharovNaorSegevShahaf'16]

Theorem:
Set **B=|X|/O(log k loglog k)** (where k is the security parameter). Then, with an overwhelming probability, the maximal load is **Z=3logk loglogk**

- This yields a **Non-Recurrent File Hashing Scheme** with:
  - Space: **B x Z = O(|X|)**
  - Locality (Search): **O(1)**
  - Bandwidth: **Õ(log k)**
- How to perform **Build(X)** obliviously?

Implementing Build Obliviously Using Locality-Friendly Oblivious-Sort

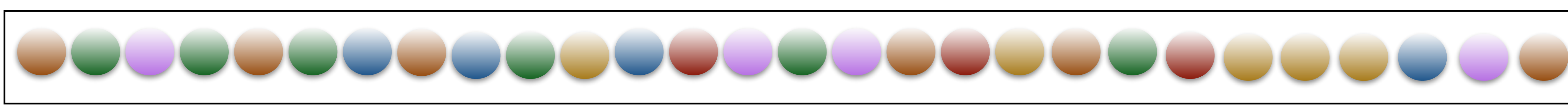
Input: Array **X**. Each element of the format (`fid,offset,data`)



Build

Input: Array **X**. Each element of the format (`fid,offset,data`)

- Choose a random PRF key **K**
- Assign to each element its dest bin: **PRF_K(`fid`)+offset** **Oblivious Sort?**
- Add **ZB** new dummy elements (doubles the structure)
    - Assign **Z** dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements

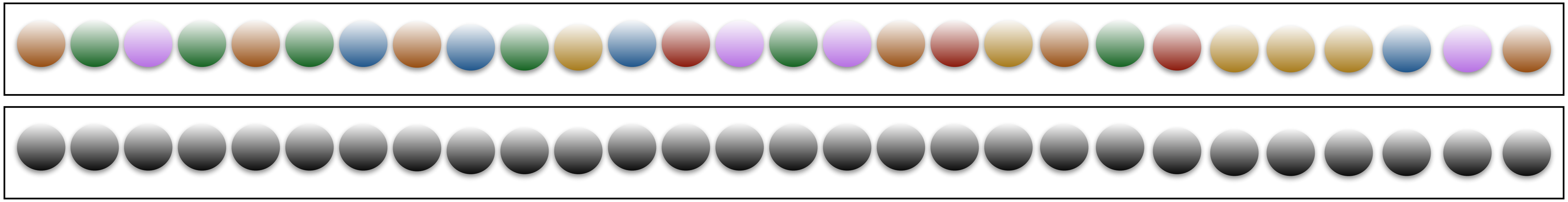


Build

Input: Array **X**. Each element of the format (`fid,offset,data`)

- Choose a random PRF key **K**
- Assign to each element its dest bin: **PRF_K(`fid`)+offset**
- Add **ZB** new dummy elements (doubles the structure)
    - Assign **Z** dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements

Z=bin size
B=number of bins

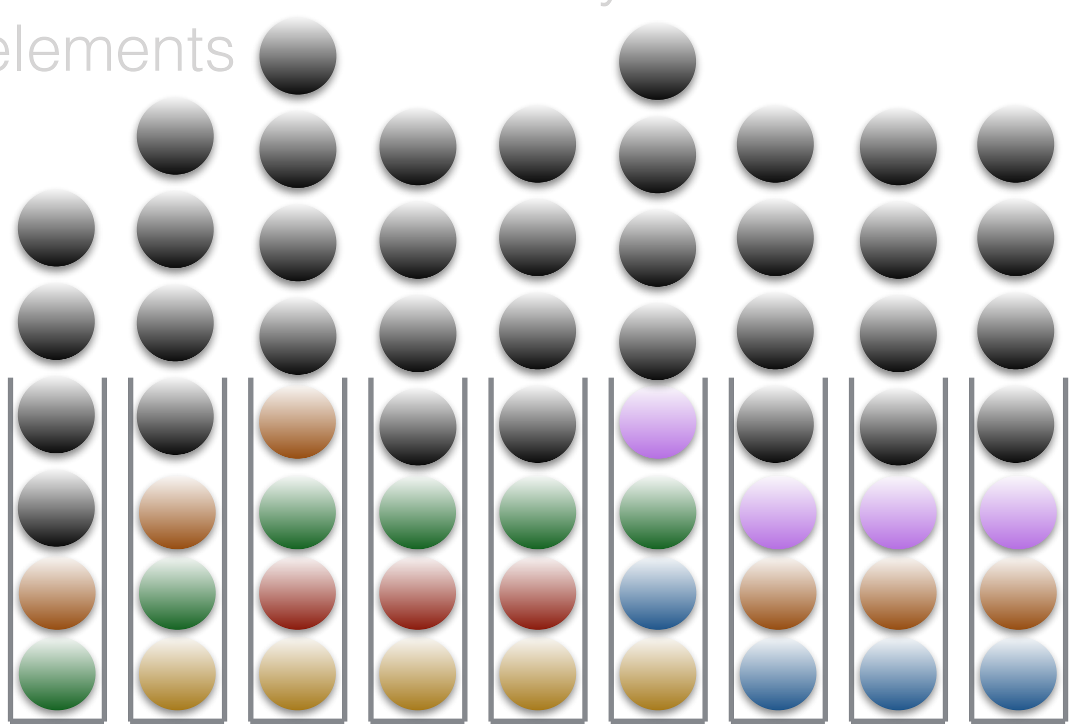


Build

Input: Array **X**. Each element of the format (`fid,offset,data`)

- Choose a random PRF key **K**
- Assign to each element its dest bin: **PRF_K(`fid`)+offset**
- Add **ZB** new dummy elements (doubles the structure)
  - Assign **Z** dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements

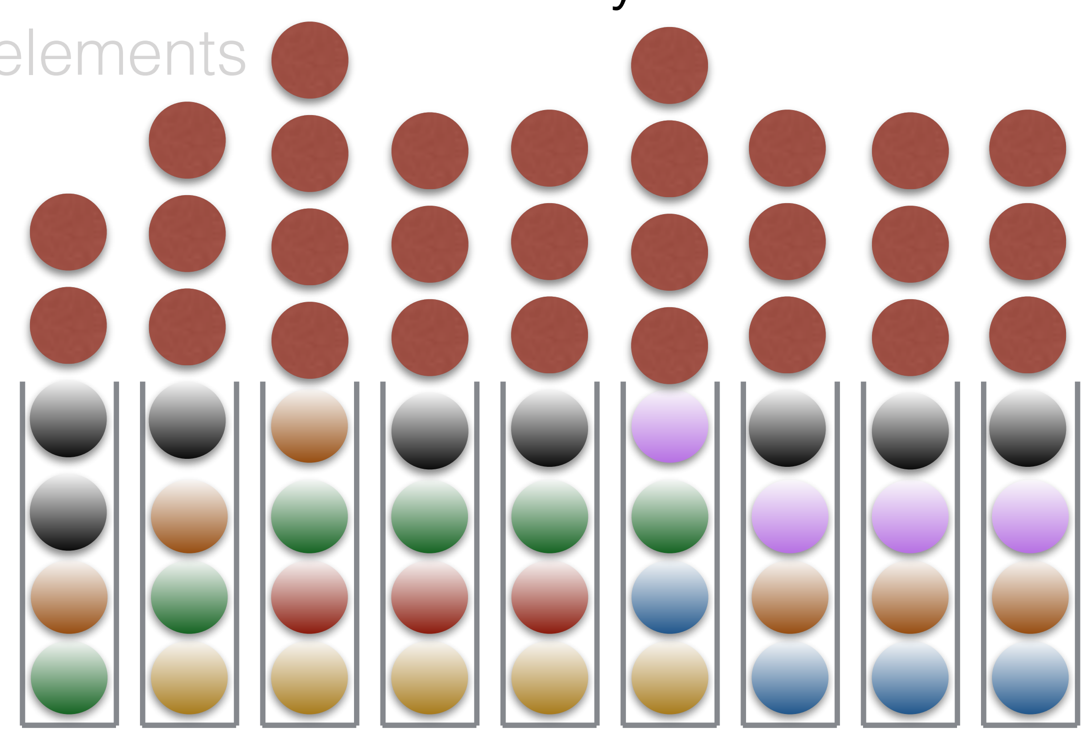
Z=bin size
B=number of bins



Build

Input: Array **X**. Each element of the format (`fid,offset,data`)

- Choose a random PRF key **K**
- Assign to each element its dest bin: **PRF_K(`fid`)+offset**
- Add **ZB** new dummy elements (doubles the structure)
  - Assign **Z** dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements

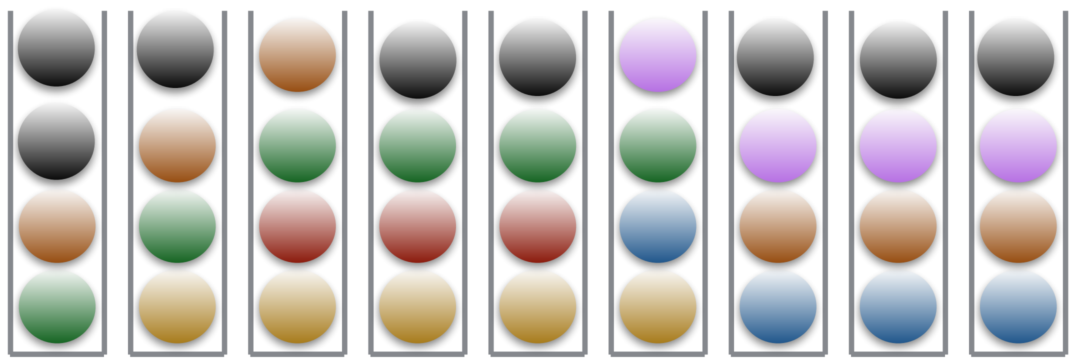


(in each bin, number of exceeded elements = number of real elements)

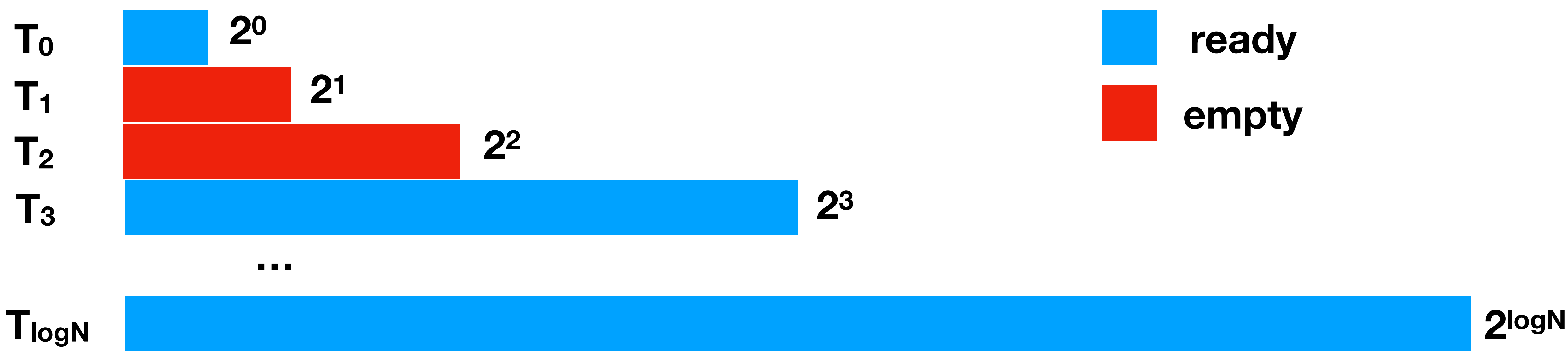
Build

Input: Array **X**. Each element of the format (`fid,offset,data`)

- Choose a random PRF key **K**
- Assign to each element its dest bin: **$\mathbf{PRF_K}$(`fid`)+offset**
- Add **ZB** new dummy elements (doubles the structure)
  - Assign **Z** dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements



File ORAM: Construction



Hierarchical construction:
Instead of a hash table in each level [GO'96]
we use non-recurrent oblivious file hashing scheme

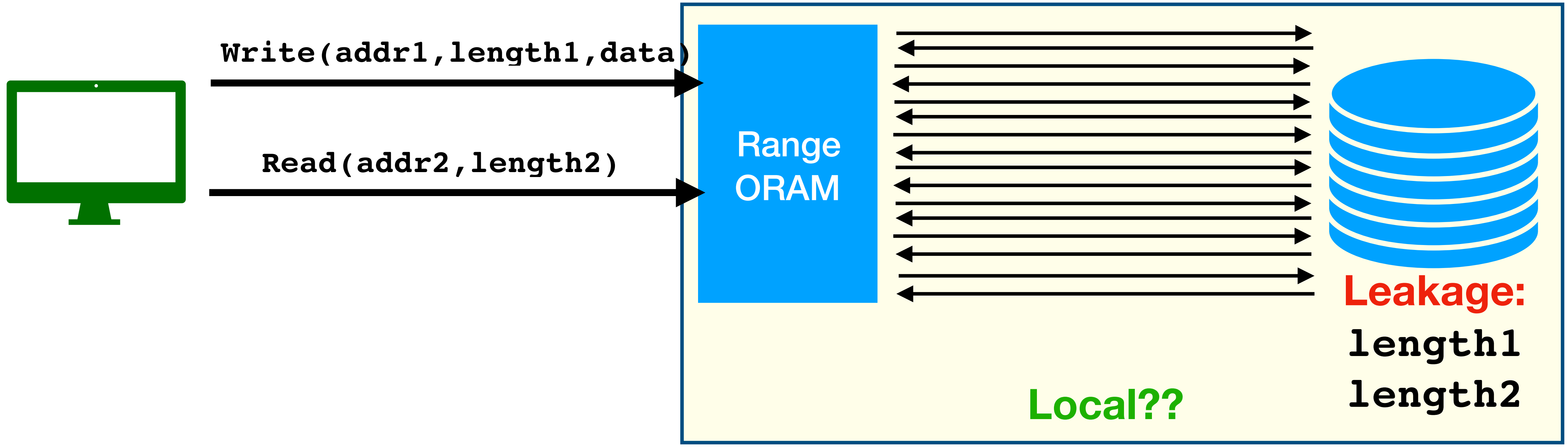
This Talk

- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	**stat**	**O(NlogN)**	**L $\tilde{O}(\log^3 N)$**	**$\tilde{O}(\log^3 N)$**	**L**
File ORAM	**comp**	**O(N)**	**L $\tilde{O}(\log^2 N)$**	**$\tilde{O}(\log N)$**	**L**
ORAM	stat	O(N)	L $o(\log^2 N)$	L $o(\log^2 N)$	none

- An intermediate result: ***Locality-Friendly oblivious sort***
  - **Perfect:** **$O(N \log^2 N)$**-work and **$O(\log^2 N)$**-locality
  - **Statistical:** **$\tilde{O}(N \log N)$**-work and **$\tilde{O}(\log N)$**-locality

First Primitive:
Range ORAM



Read Only Range ORAM

ORAM $B=1$ | ORAM $B=2^1$ | ORAM $B=2^2$ | ... | ORAM $B=2^{\log N}$

- Store multiple copies of the data
  - **logN** ORAMs, each based on a different block-size B
- `Read(addr,2^i)` - fetches 2 blocks from the *i*th ORAM
  - Leaks $L=2^i$
- Space: $O(N\log N)$, Bandwidth: $o(L\log^2 N)$, locality $o(L\log^2 N)$

But.. what should we do with writes?

```
Write(31,data,1)          Read(16,data,64)
Write(17,data,1)
```

Range ORAM

- Range Trees
- Dealing with multiple copies of the data
  - Data coherency
- Extensions: Online Range Data
- Perfect Security

This Talk

- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	stat	$O(N\log N)$	$L\ \tilde{O}(\log^3 N)$	$\tilde{O}(\log^3 N)$	L
File ORAM	comp	$O(N)$	$L\ \tilde{O}(\log^2 N)$	$\tilde{O}(\log N)$	L
ORAM	stat	$O(N)$	$L\ o(\log^2 N)$	$L\ o(\log^2 N)$	none

- An intermediate result: ***Locality-Friendly oblivious sort***
  - **Perfect: $O(N \log^2 N)$-work and $O(\log^2 N)$-locality**
  - **Statistical: $\tilde{O}(N \log N)$-work and $\tilde{O}(\log N)$-locality**

Oblivious Sorting

- Tremendous amount of applications…
- Asymptotically best known oblivious sorts are **O(nlogn) work** (but not locality-friendly)
  - AKS (1983) - based on expanders, theoretical
  - ZigZag sort (Goodrich, STOC'14)
    - Very large constants..
  - Randomized Shell Sort [Goodrich'11] — not local
- **In practice:** Batcher (1968) — O(n $\log^2$n)
  - Good locality, (perfect!) — not asymptotically optimal
- If we want **Range ORAM** and **File ORAM** with efficiency comparable to ordinary ORAM — we need **a better oblivious sort**

Locality-Friendly Oblivious Sort

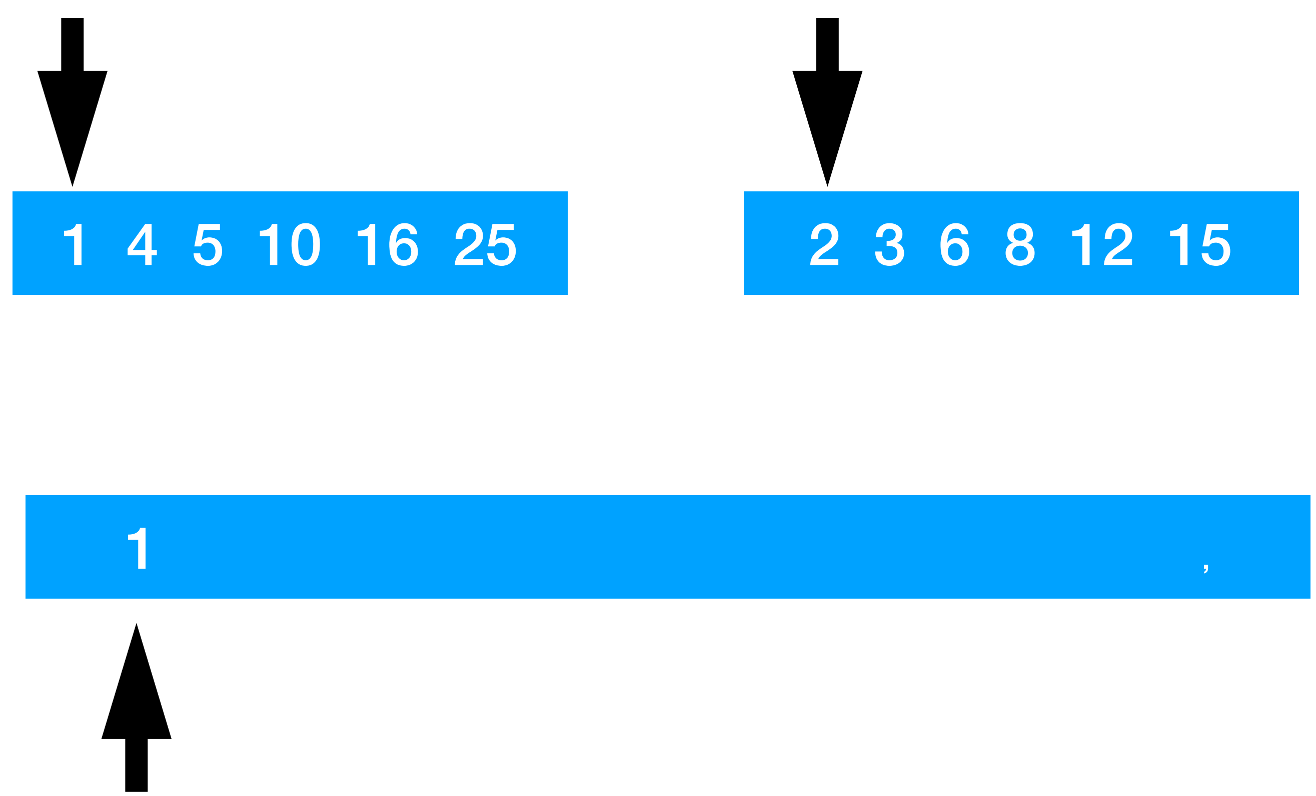
	Oblivious	Local	Complexity
Merge Sort			
Bitonic Sort			
Our Sort			

Merge Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort			
Our Sort			

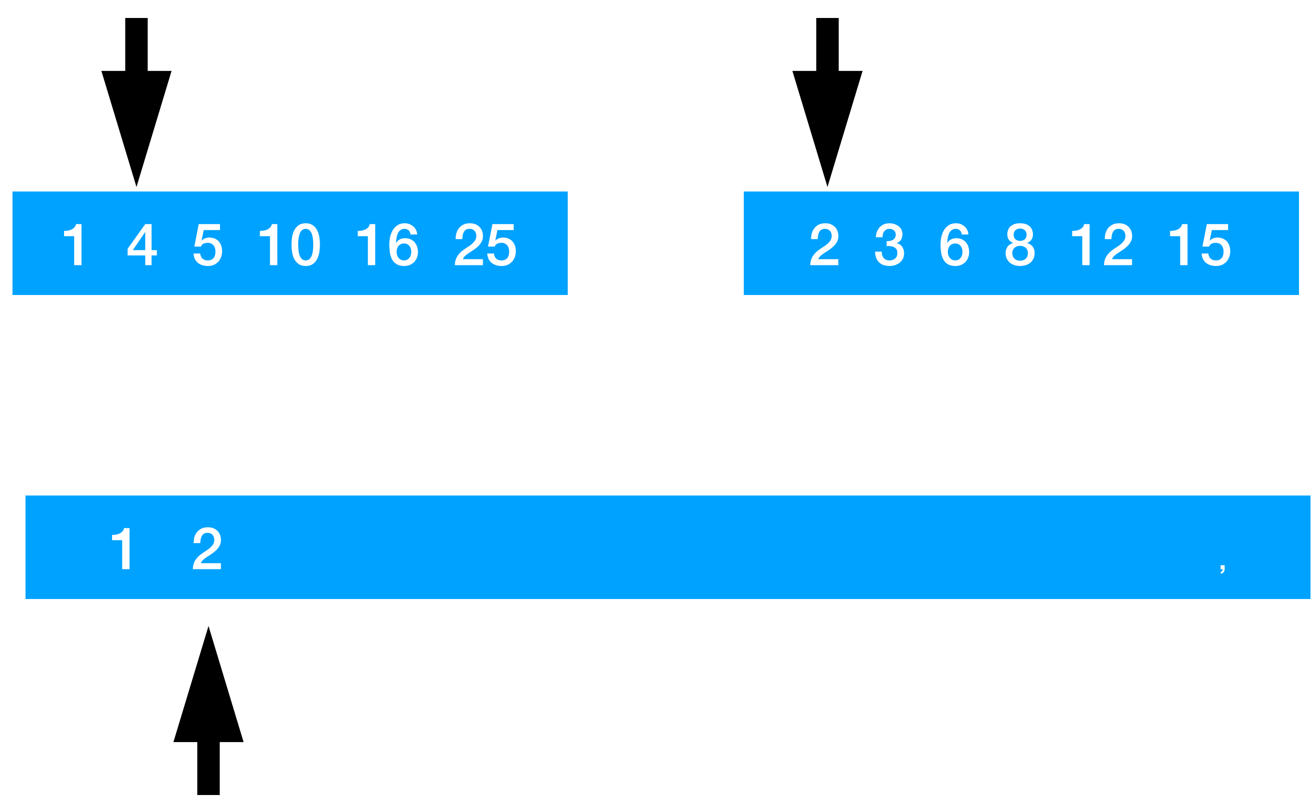
Merge Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort			
Our Sort			



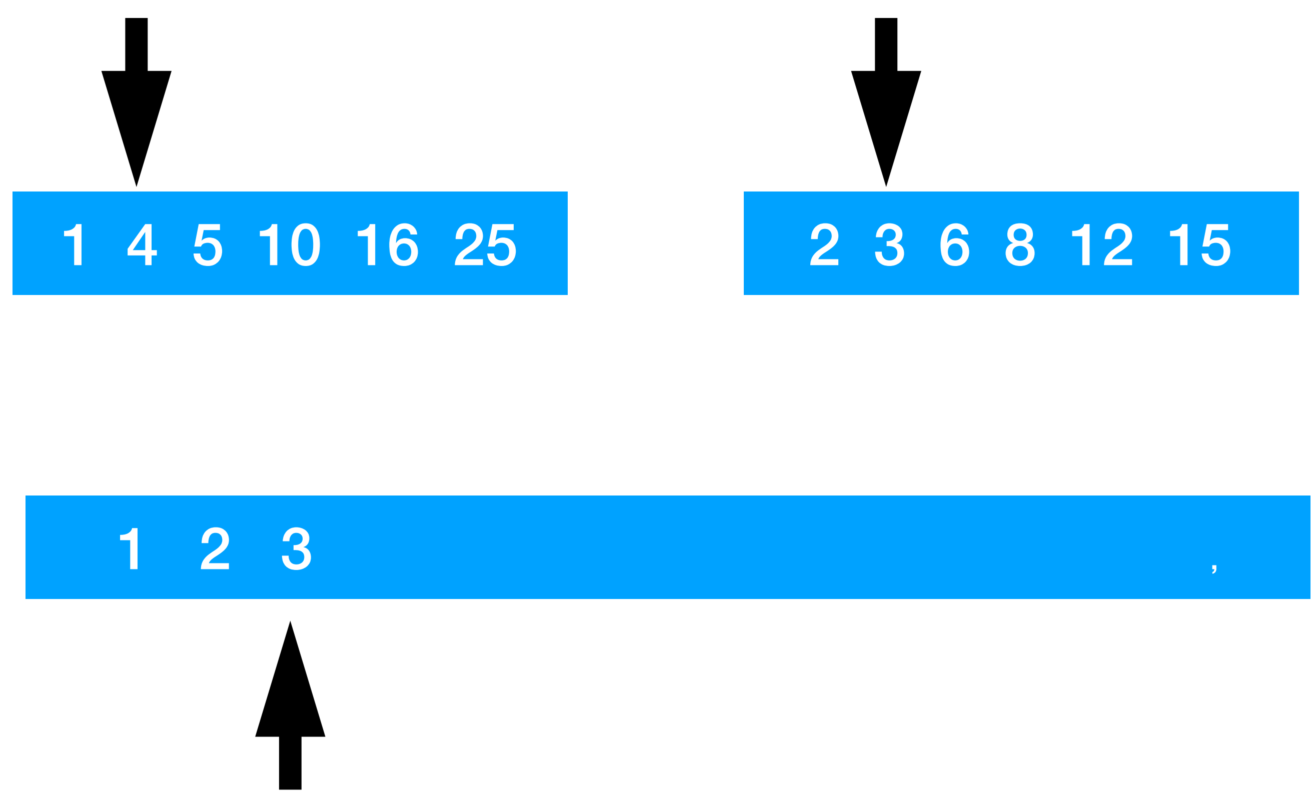
Merge Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort			
Our Sort			



Merge Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort			
Our Sort			



Merge Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort			
Our Sort			

1 4 5 10 16 25

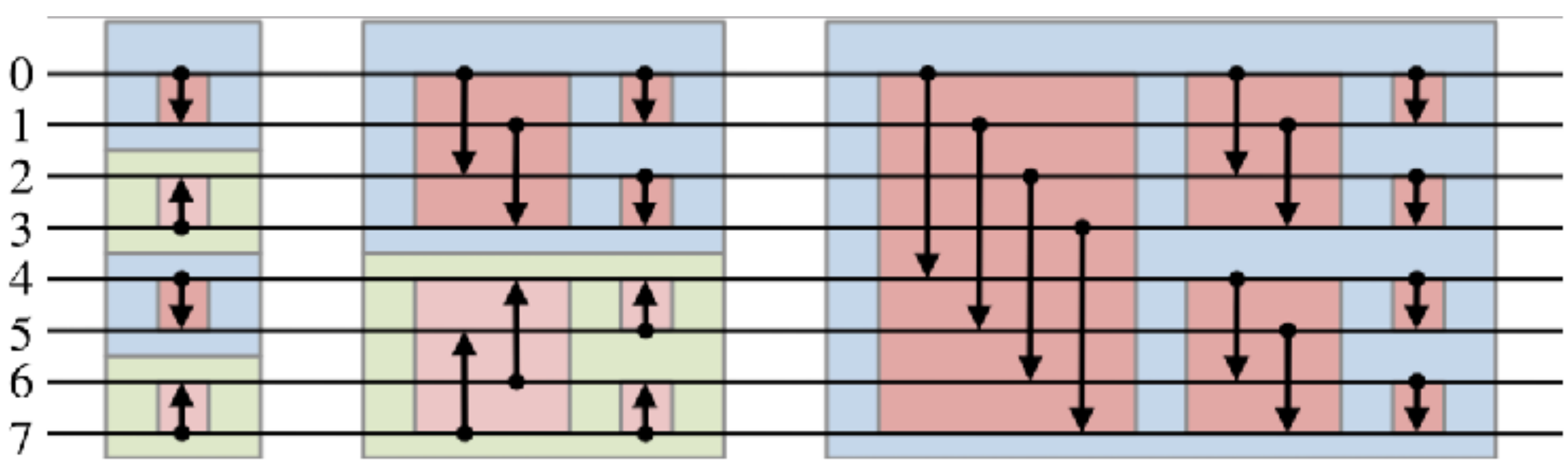
2 3 6 8 12 15

1 2 3 4 5 6 8 10 12 15 16 25

Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	O(Nlog²N)
Our Sort			

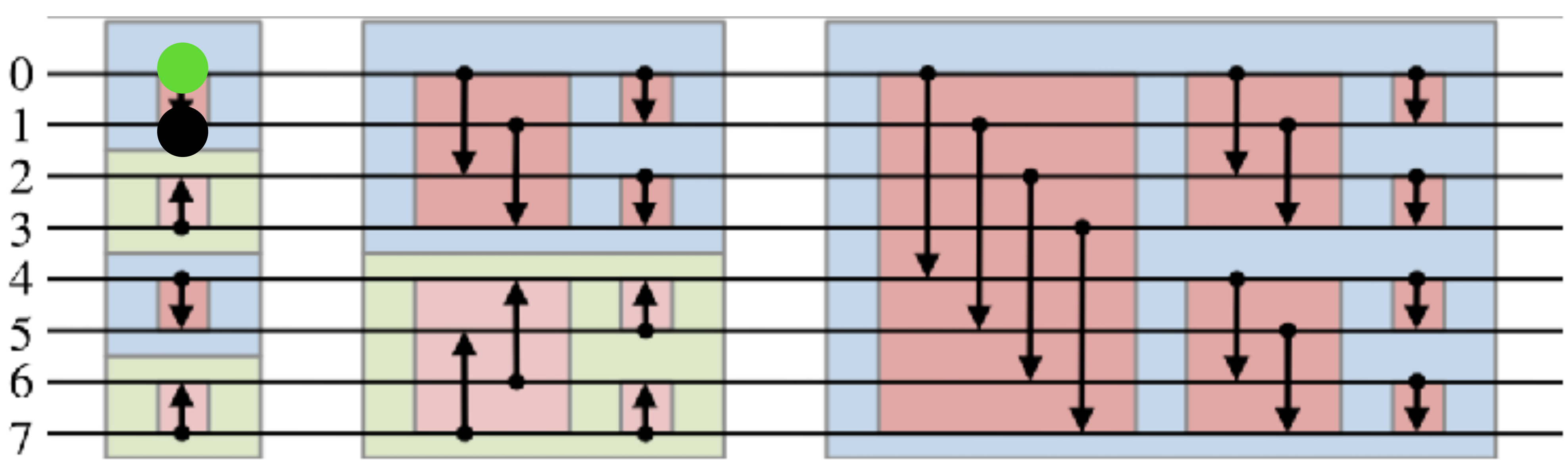
- Batcher sorting:



Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	$O(N\log^2 N)$
Our Sort			

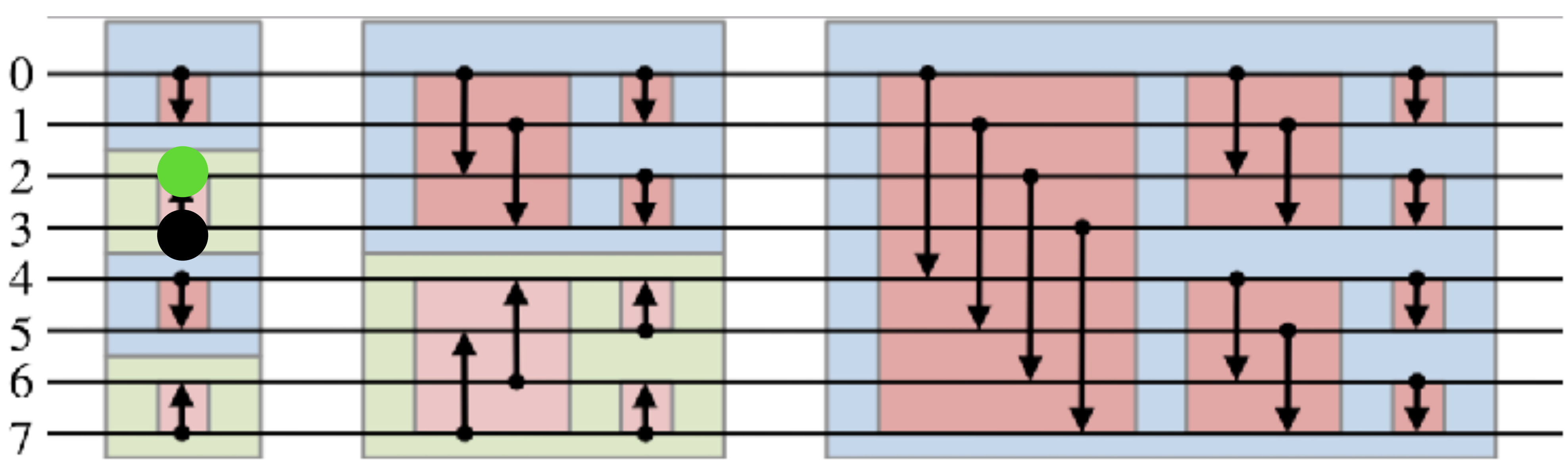
- Batcher sorting:



Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	O(Nlog²N)
Our Sort			

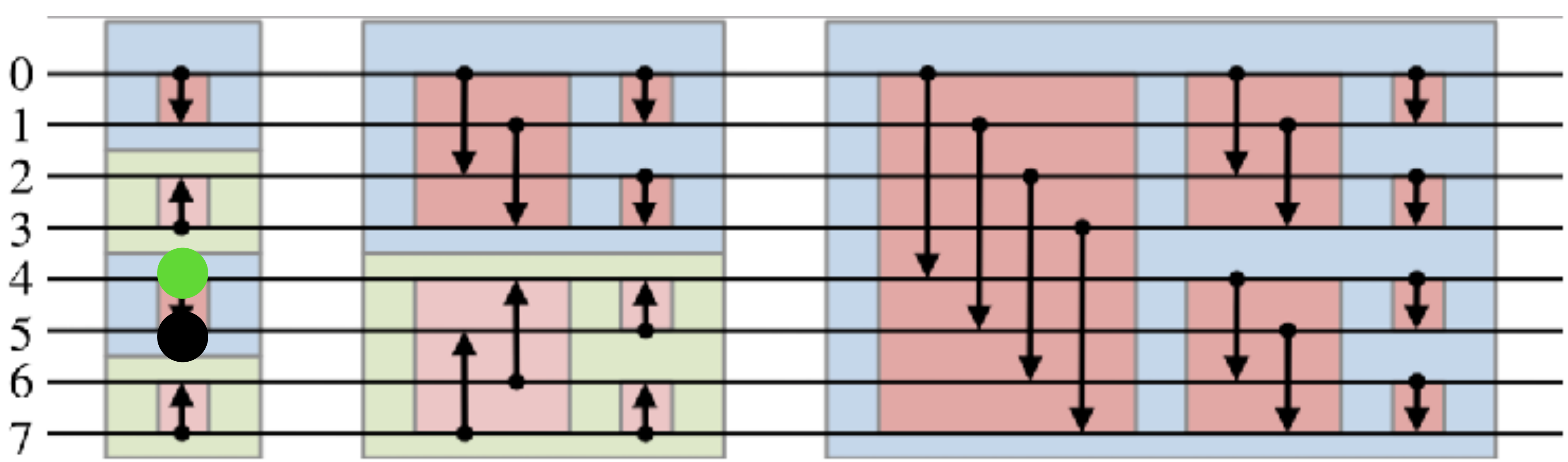
- Batcher sorting:



Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	O(Nlog²N)
Our Sort			

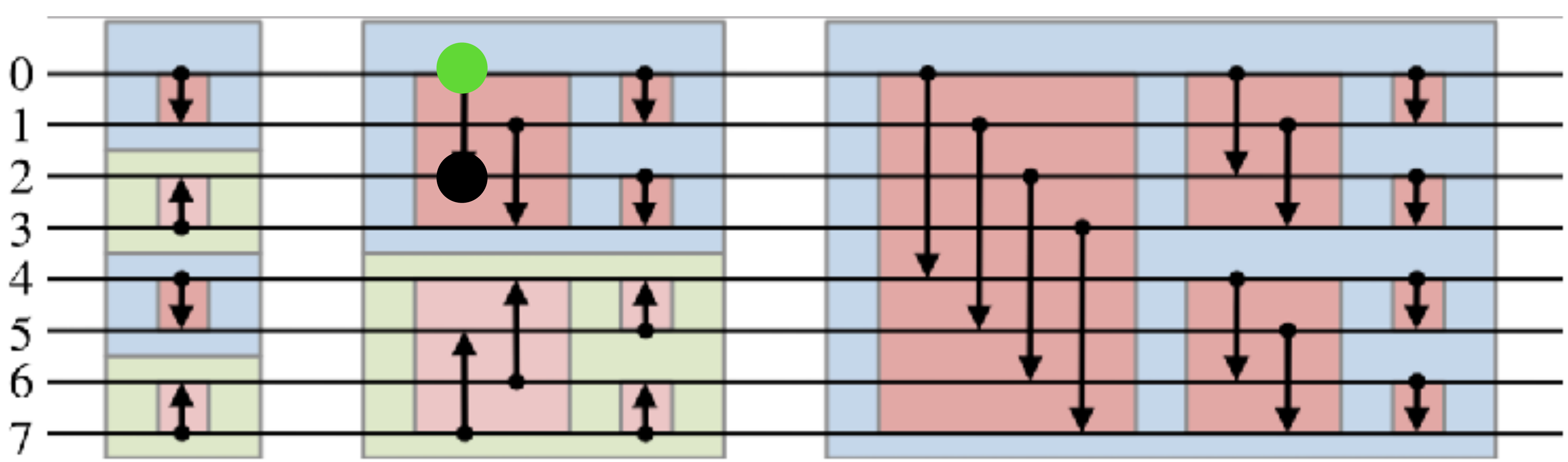
- Batcher sorting:



Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	O(Nlog²N)
Our Sort			

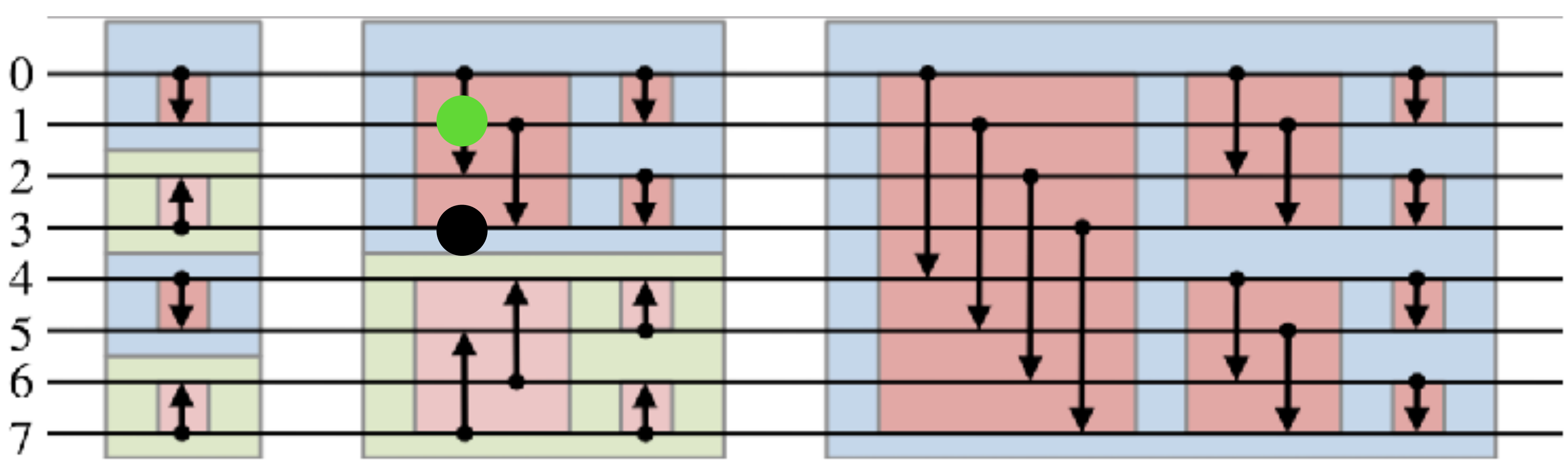
- Batcher sorting:



Bitonic Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔	O(NlogN)
Bitonic Sort	✔	✔	$O(N\log^2 N)$
Our Sort			

- Batcher sorting:

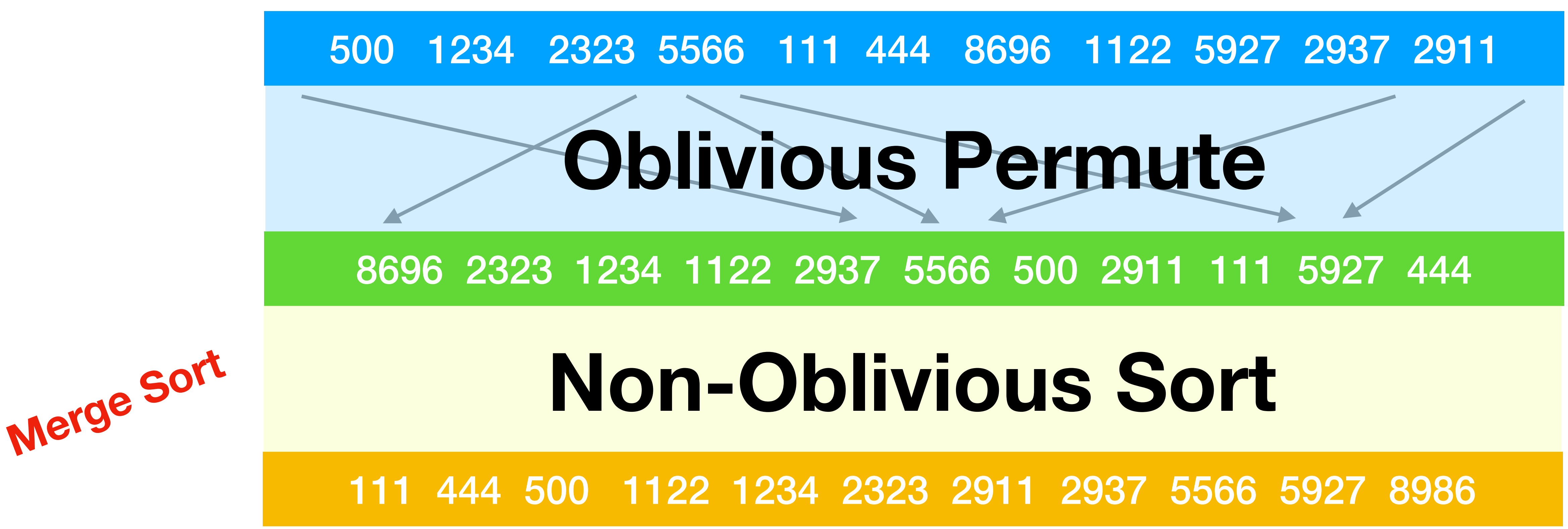


Our Sort

	Oblivious	Local	Complexity
Merge Sort	✗	✔ (3 heads)	O(NlogN)
Bitonic Sort	✔	✔ (2 heads)	$O(N\log^2 N)$
Our Sort	✔	✔ (3 heads)	$O(N\log N \log\log^2 k)$

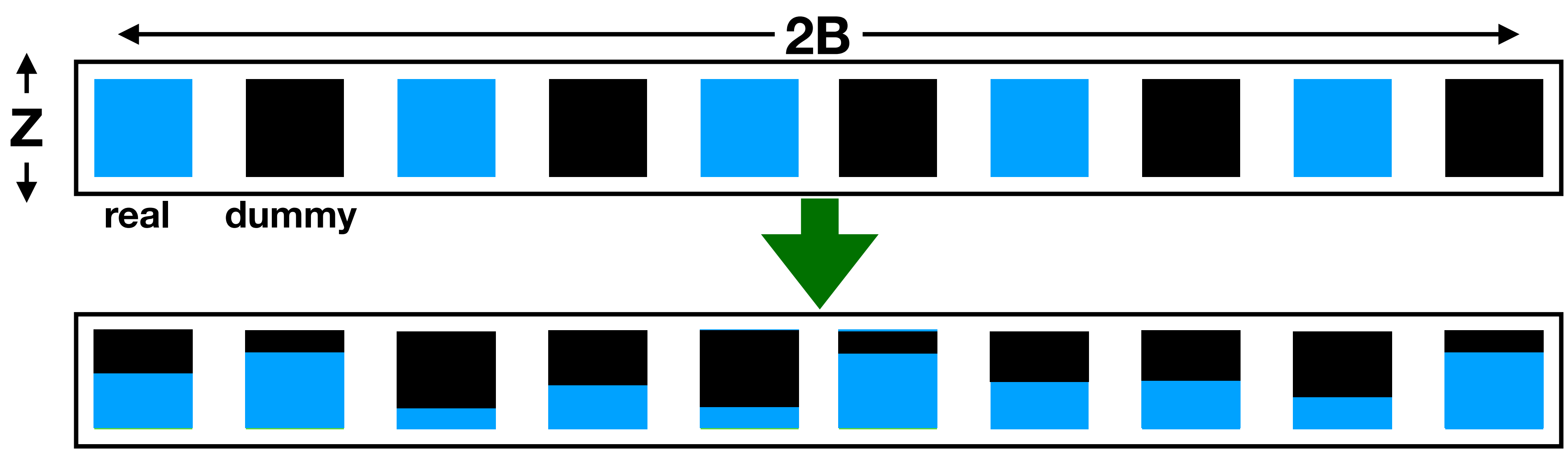
Perfect security

Statistical security



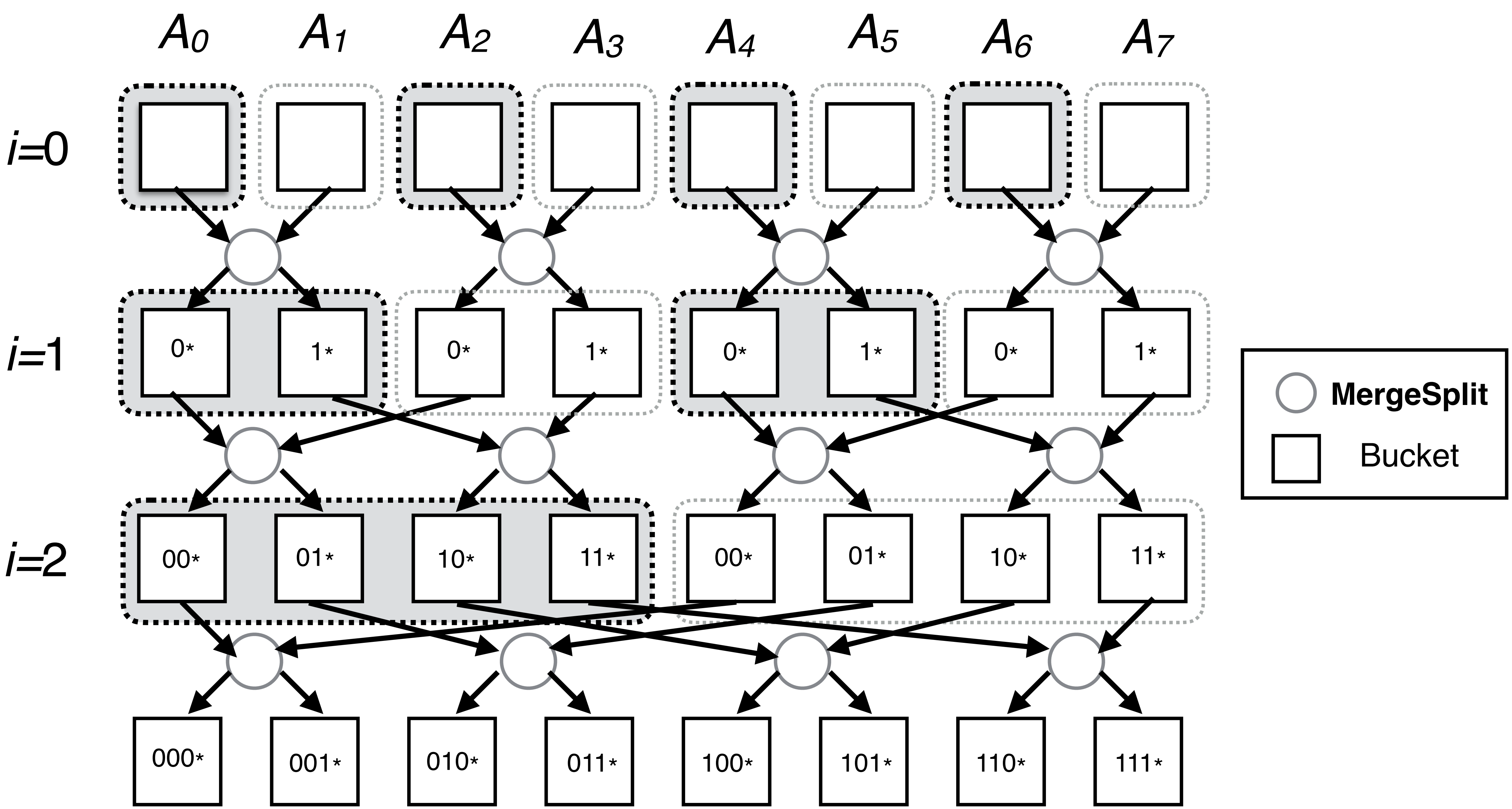
Our Oblivious Permute

- We show how to implement oblivious permutation with "slack"
  - introducing some "dummy" values between real-values
- Interpret the input array as **B** buckets of size **Z** each
  (**Z**=**poly** log k, **B**=N/**Z**, k is the security parameter)
  - Add a bucket of dummy elements between two "real" buckets
  - Assign to each element a *random* destination bin [1,…,B]

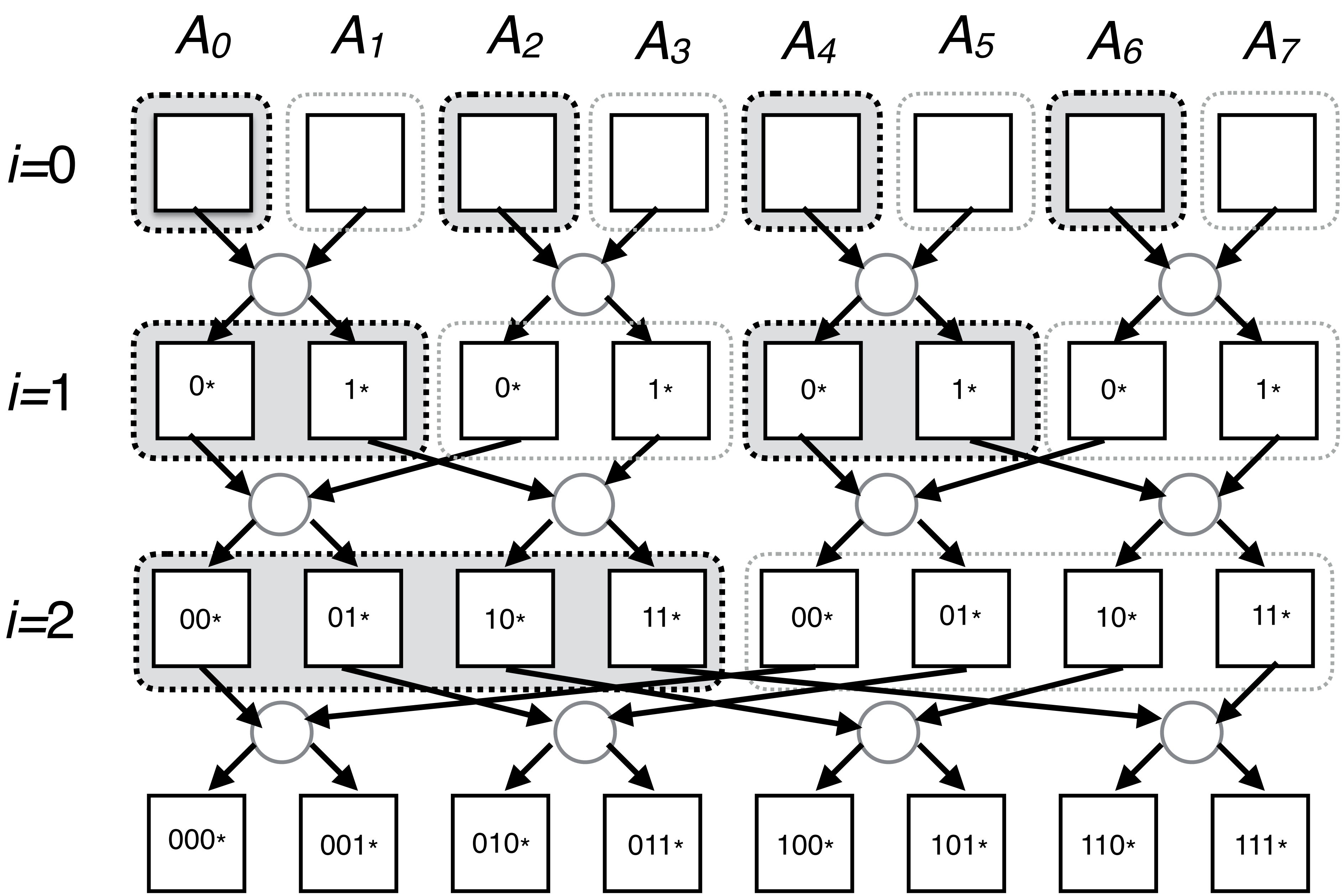


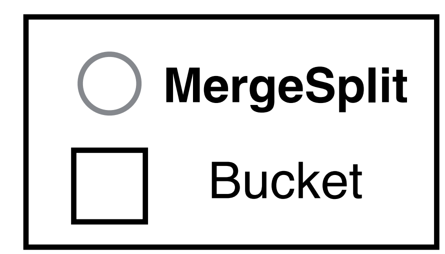
(We later remove these dummy elements using the non-oblivious sort)

Our Oblivious Permute



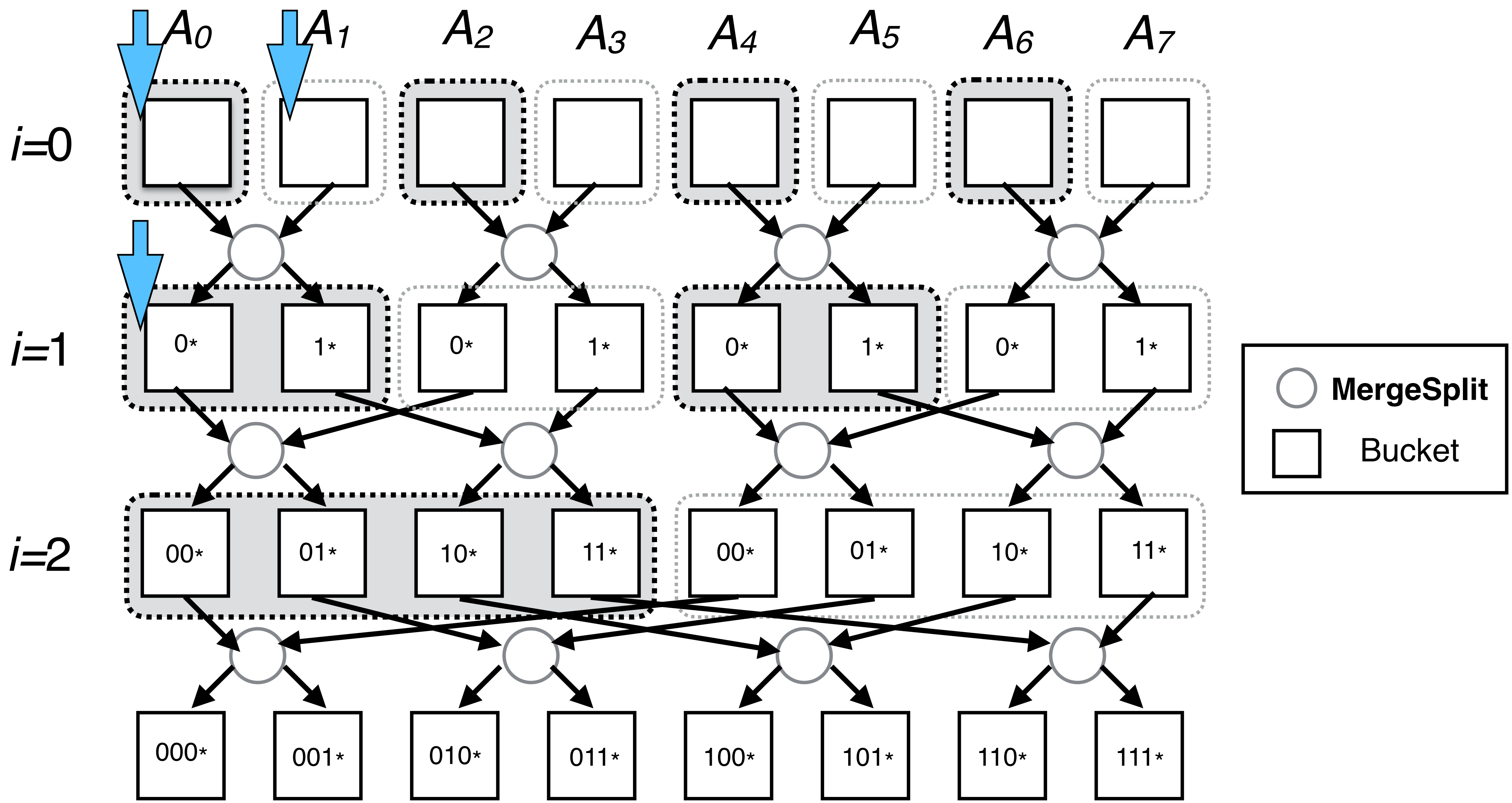
Our Oblivious Permute



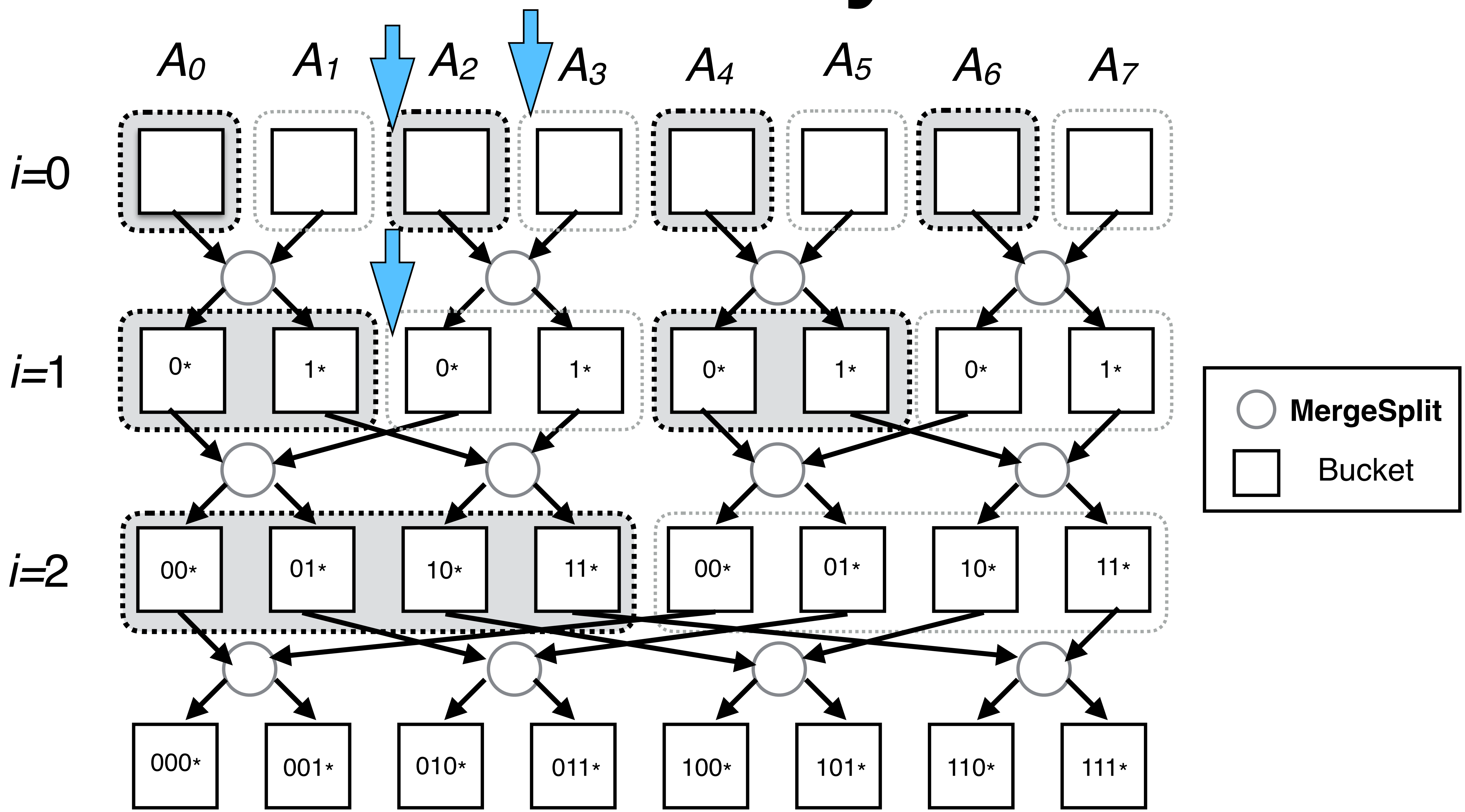


MergeSplit - takes all read elements in input buckets and distribute them to output buckets according to the i^{th} MSB

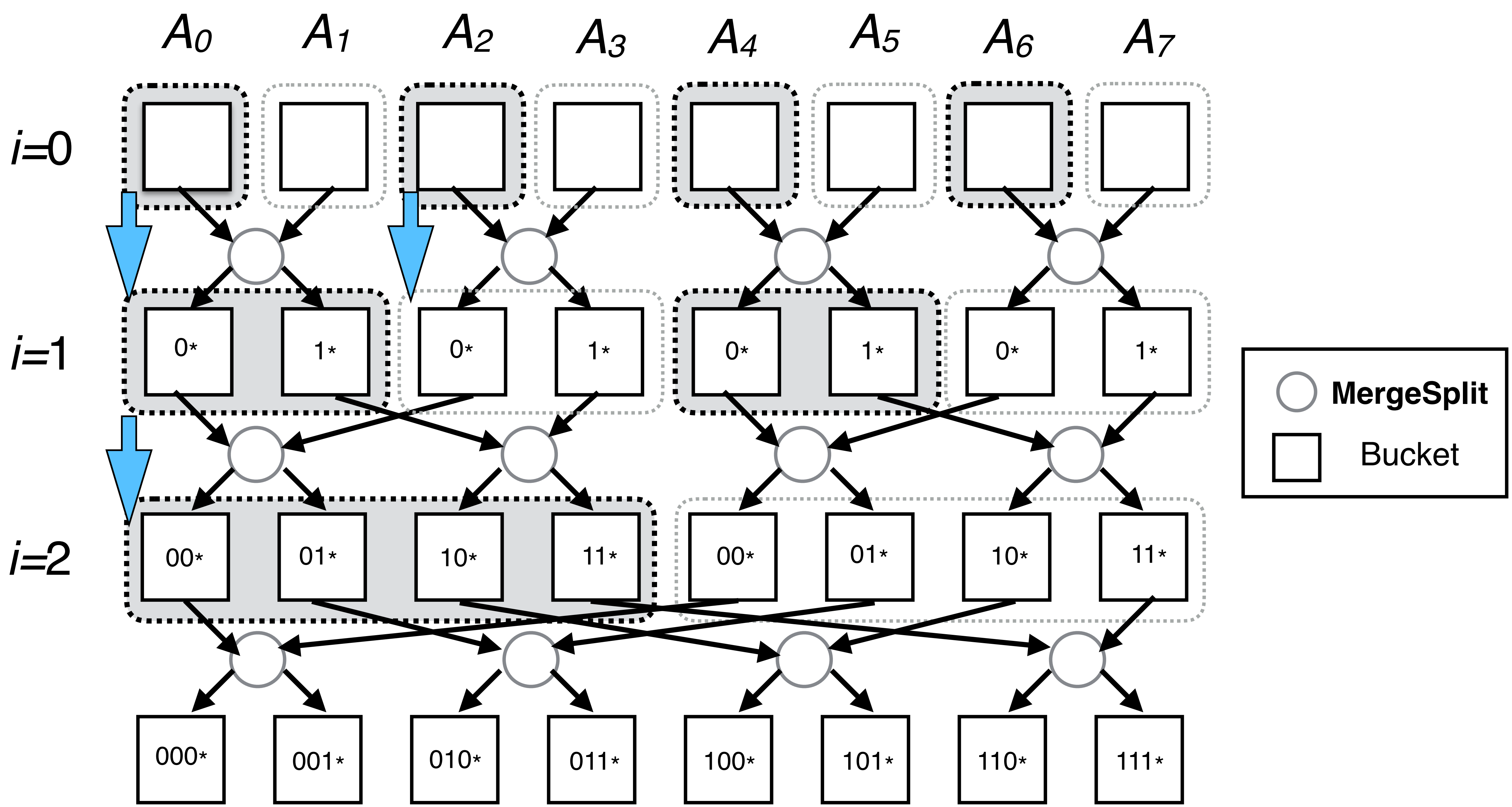
Our Oblivious Permute - Locality



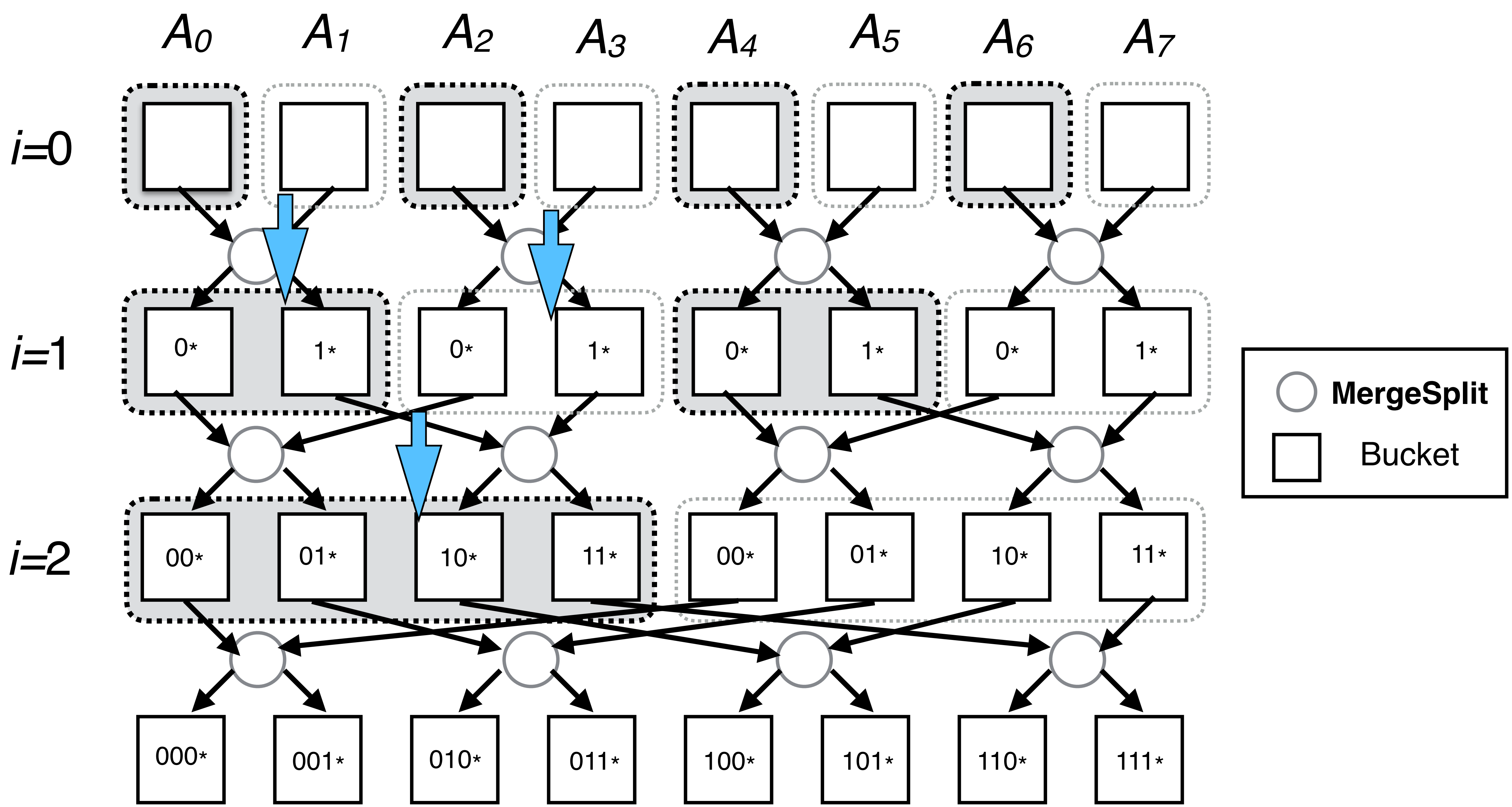
Our Oblivious Permute - Locality



Our Oblivious Permute - Locality



Our Oblivious Permute - Locality



Are We Done?

- **Claim**: for *random assignment* of destination buckets, *overflow* with only *negligible* probability
- However this is not a permutation!
  - The buckets are not permuted…
- We obliviously sort (bitonic sort) each *bucket* according to the final assignment
    - $BZ\log^2 Z = n/\log k * \log k * \log^2\log k = n \log^2\log k$
  - Not a permutation, but composition works
- There is an easier solution if the CPU has non-constant size

Theorem:

There exists a *statistically* secure **oblivious sort** algorithm that completes in $O(n \log n \log\log^2 k)$ work and $(3, O(\log n \log\log^2 k))$-locality

Conclusions

- We introduce **locality** in oblivious RAM
- **Impossibility**: locality without leakage of lengths

	Security	Space	Bandwidth	Locality	Leakage
Range ORAM	stat	O(NlogN)	L $\tilde{O}(\log^3 N)$	$\tilde{O}(\log^3 N)$	L
File ORAM	comp	O(N)	L $\tilde{O}(\log^2 N)$	$\tilde{O}(\log N)$	L
ORAM	stat	O(N)	L $o(\log^2 N)$	L $o(\log^2 N)$	none

- An intermediate result: Locality-Friendly oblivious sorting algorithms
  - **Perfect:** **$O(N \log^2 N)$**-work and **$(2,O(\log^2 N))$**-locality
  - **Statistical: $\tilde{O}(N \log N)$**-work and **$(3,\tilde{O}(\log N))$**-locality

Thank You!