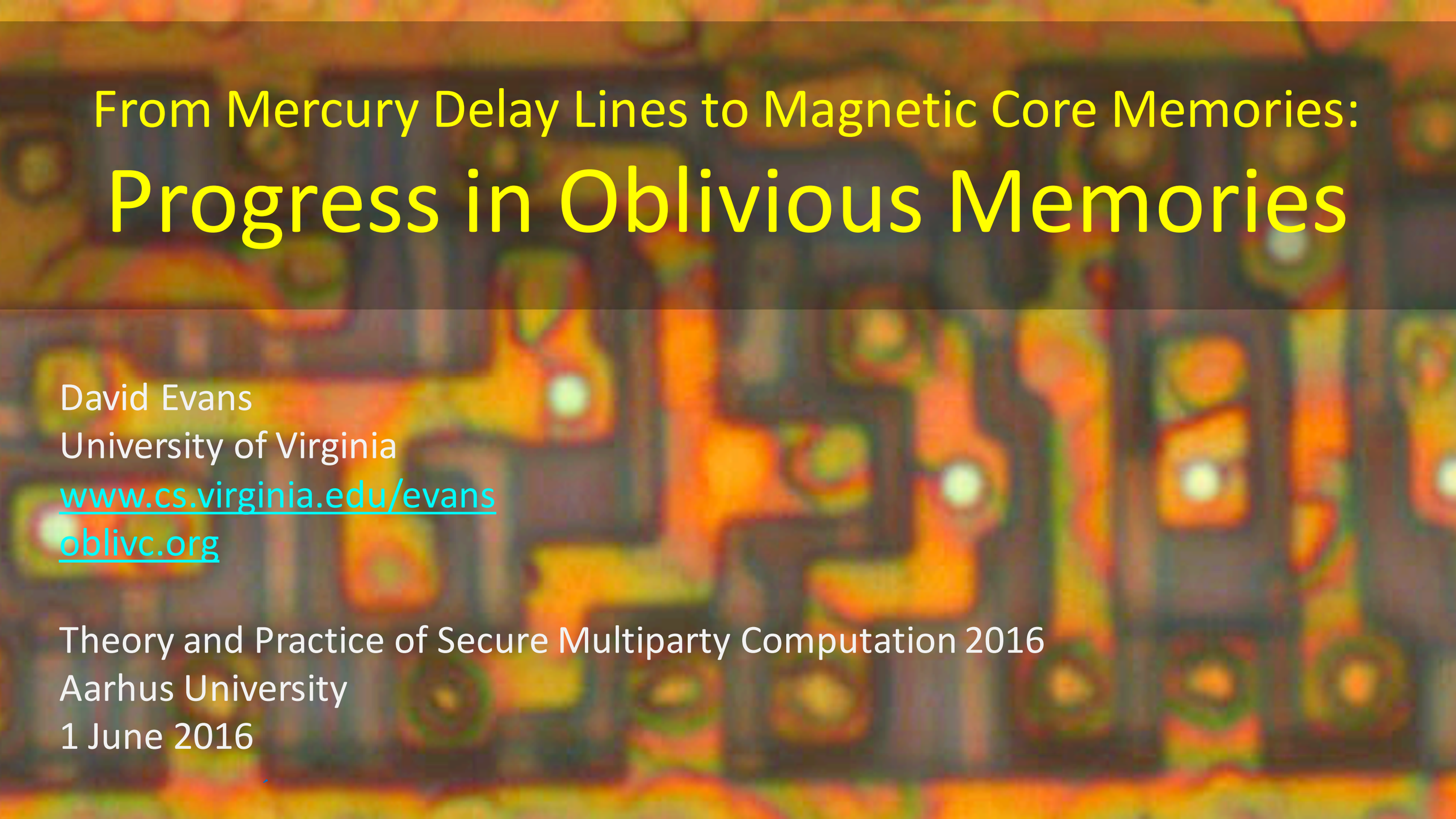


From Mercury Delay Lines to Magnetic Core Memories: Progress in Oblivious Memories

David Evans
University of Virginia
www.cs.virginia.edu/evans
oblivc.org

Theory and Practice of Secure Multiparty Computation 2016
Aarhus University
1 June 2016

Building MPC Applications

Application-Specific **General Purpose**

Custom Protocols → **Generic Protocols** (e.g., Yao's)

Custom Data Structures → **Library Data Structures**

General-Purpose ORAM

Data-Oblivious Algorithms → **Standard Algorithms**

Setting

Semi-honest model

Two-party computation

Mostly standard assumptions

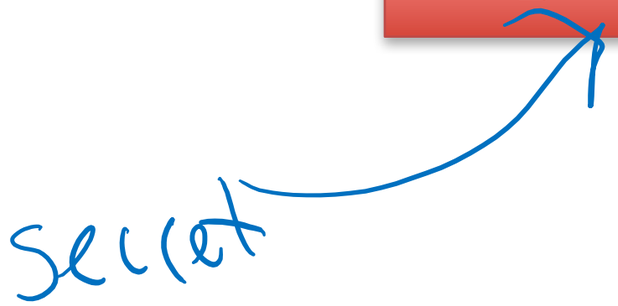
(although implementation uses Free-XOR)

Oblivious Data Structures

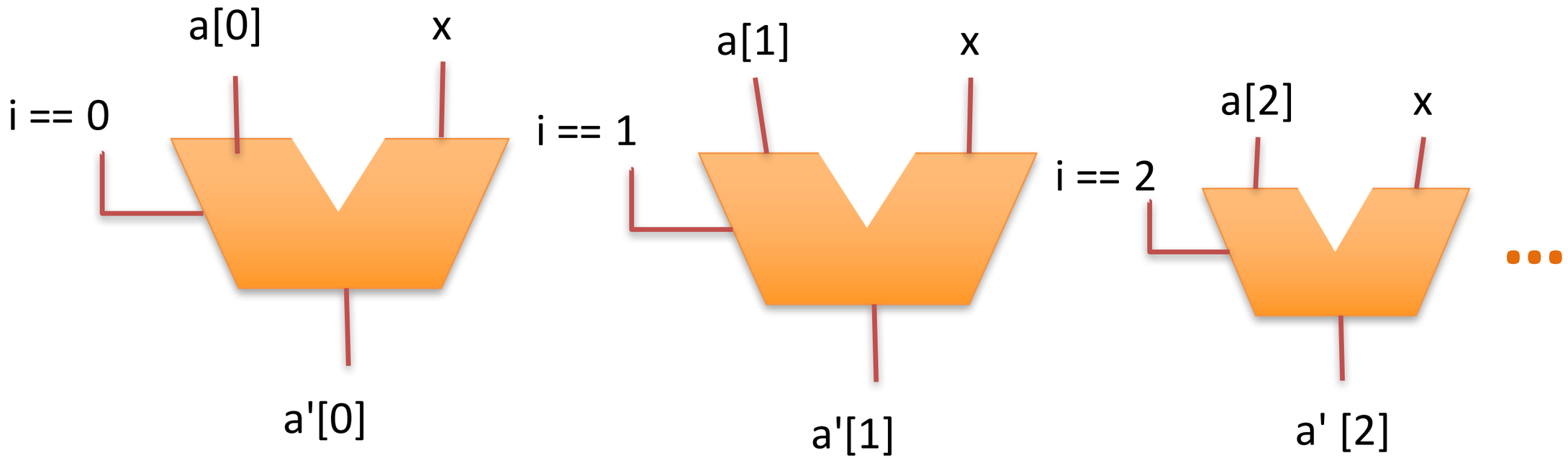
Samee Zahur and David Evans. *Circuit Structures for Improving Efficiency of Security & Privacy Tools.* **IEEE Security and Privacy (Oakland) 2013.**

Crazy Things in Typical Code

```
a[i] = x
```

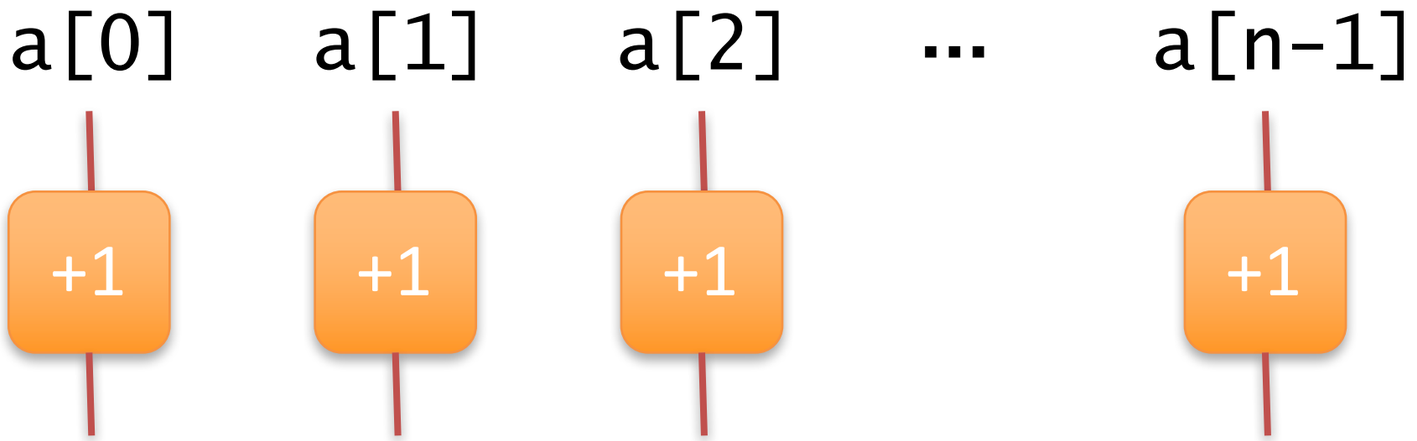


Circuit for Array Update



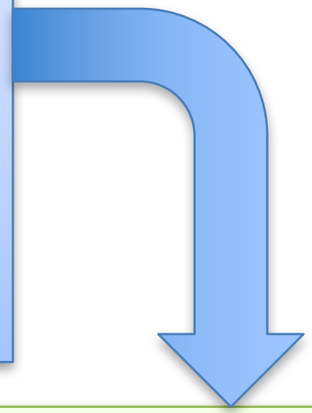
Easy (and Common) Case

```
for (i = 0; i < n; i++)
    a[i] += 1
```



Locality: Stacks and Queues

```
if (x != 0)
    a[i] += 1
    if (a[i] > 10)
        i += 1
    a[i] = 5
```

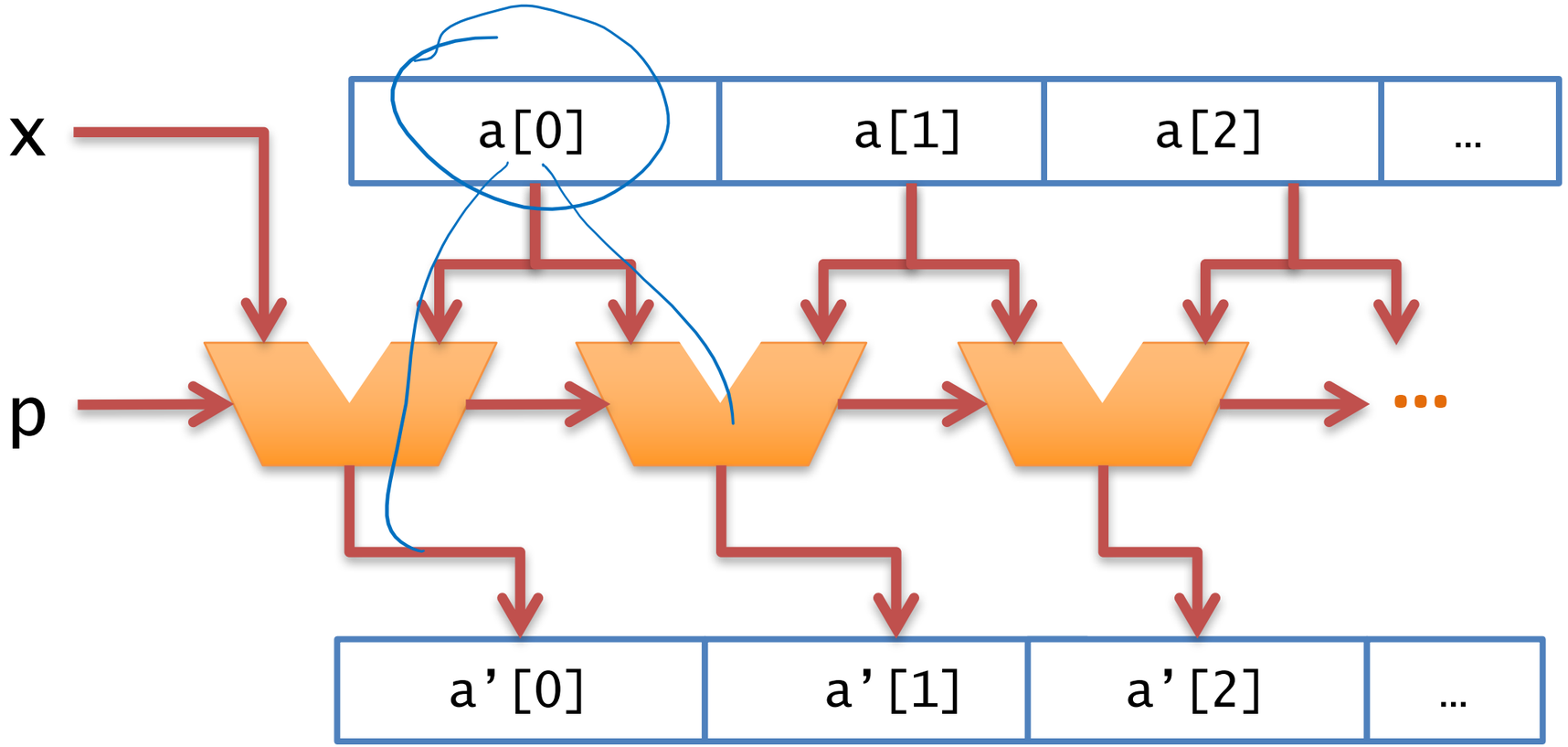


Data-oblivious code

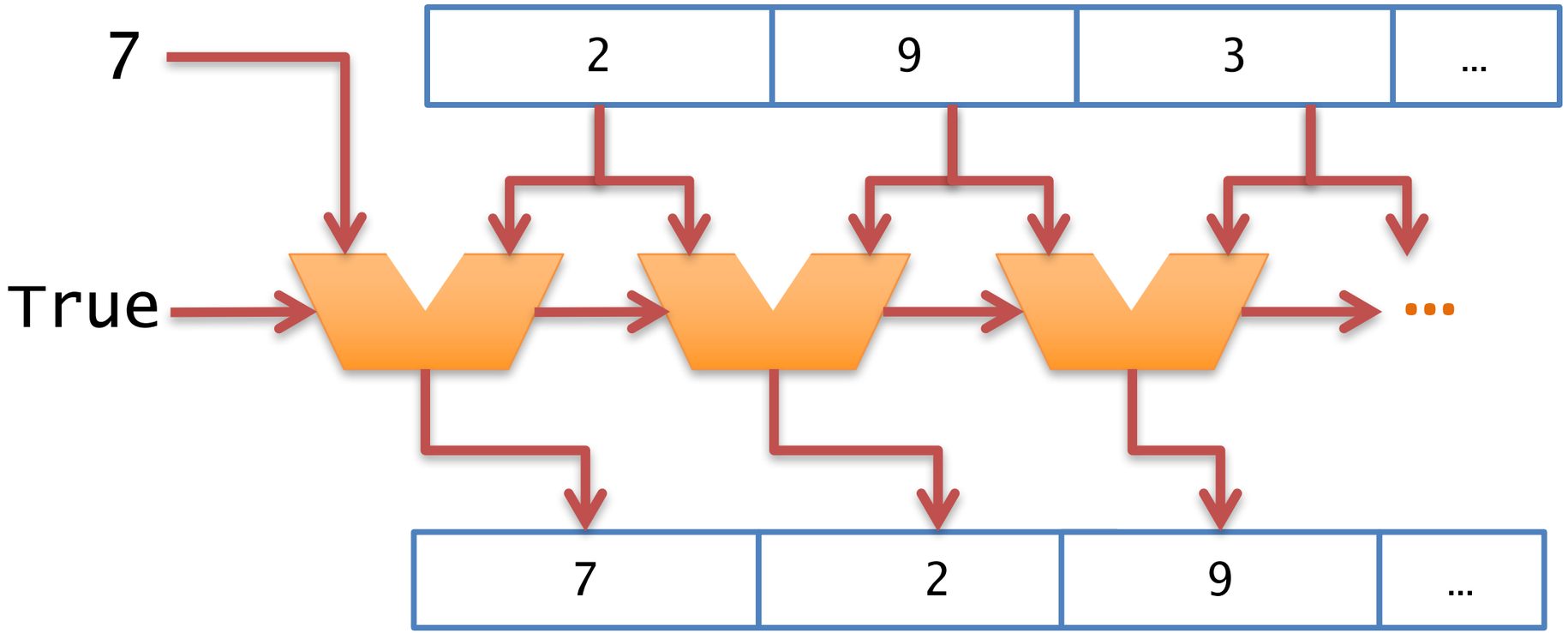
No branching allowed

```
t := a.top() + 1
a.cond_update(x != 0, t)
a.cond_push(x != 0 && t > 10, *)
a.cond_update(x != 0, 5)
```

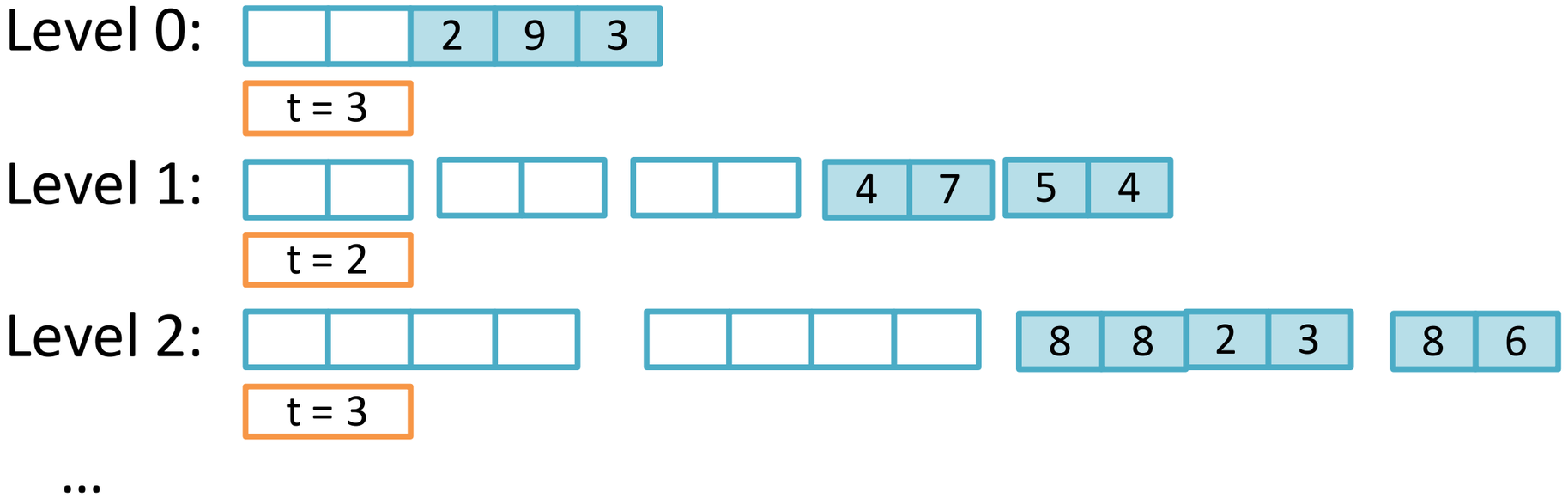
Naïve Conditional Push



Naïve Conditional Push



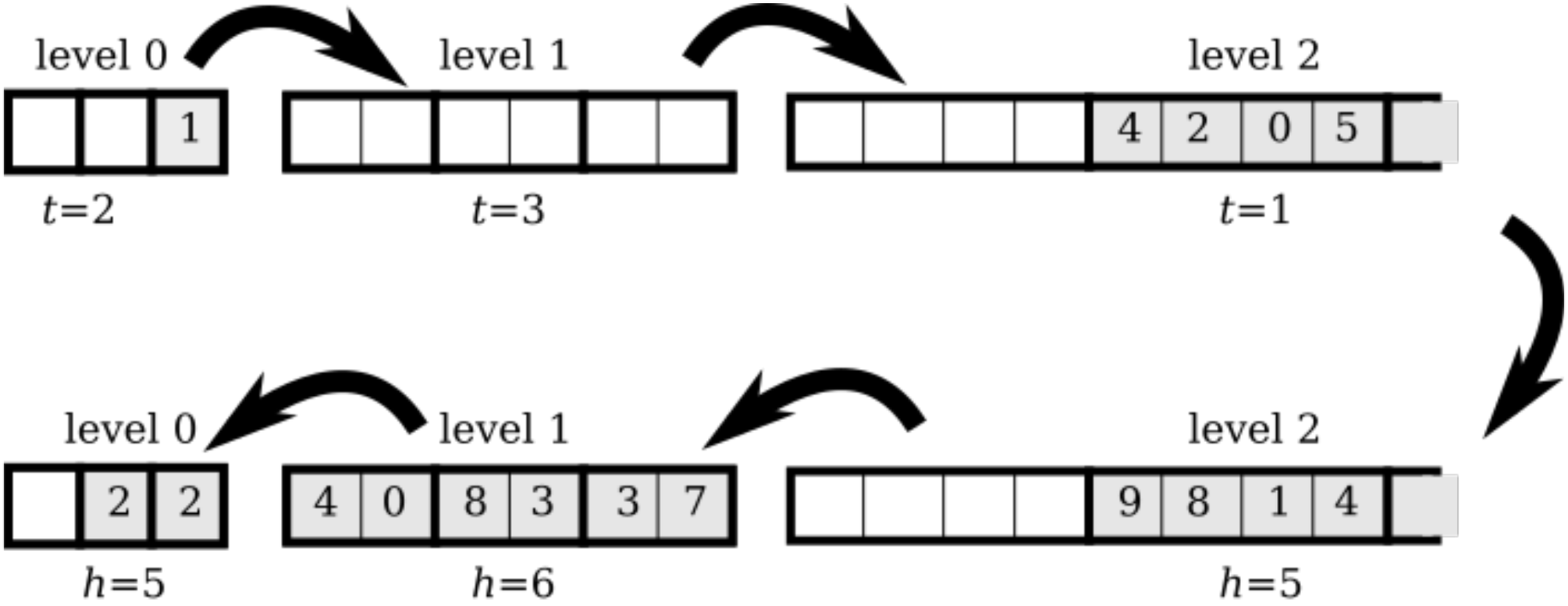
More Efficient Stack



Block size = 2^{level}

Each level has 5 blocks, at least 2 full and 2 empty

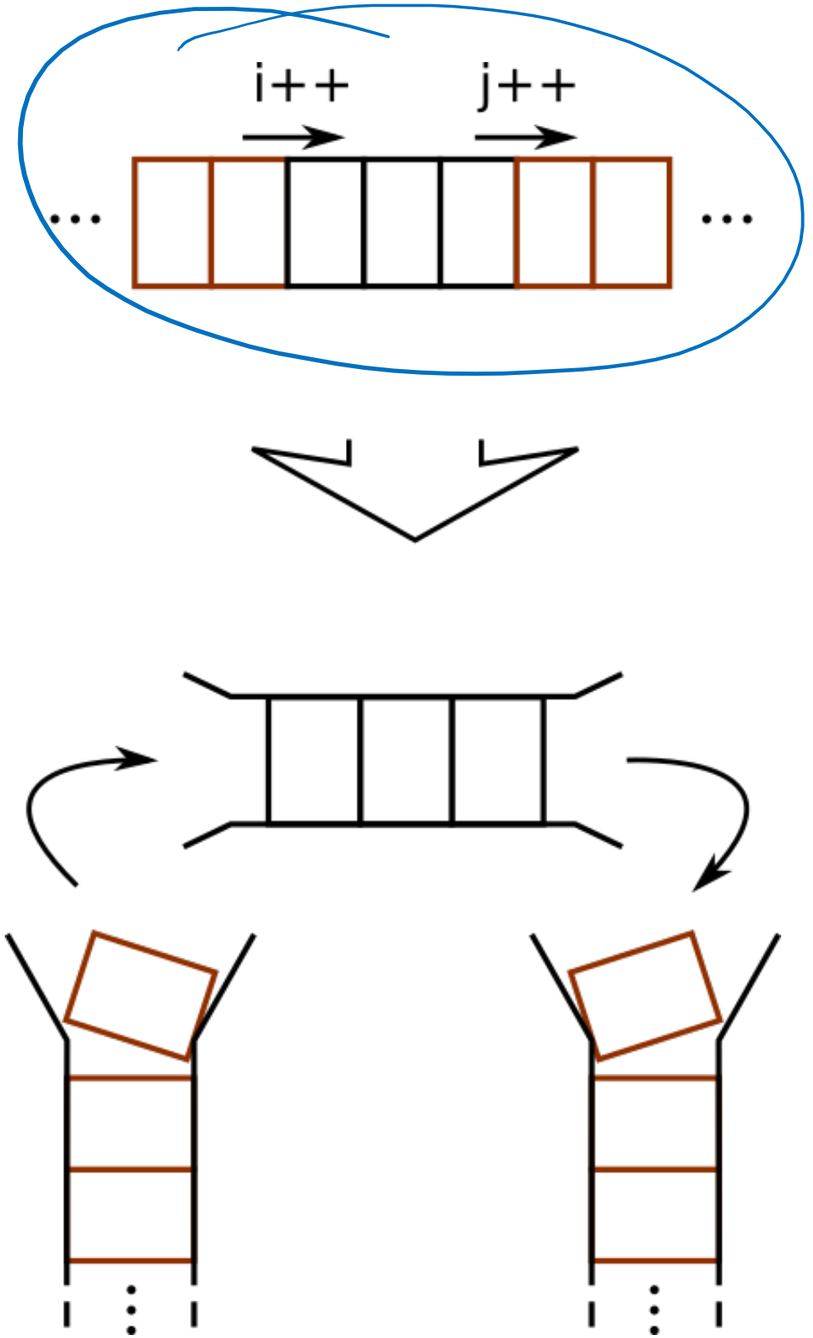
Efficient queue operations



Spatial Locality

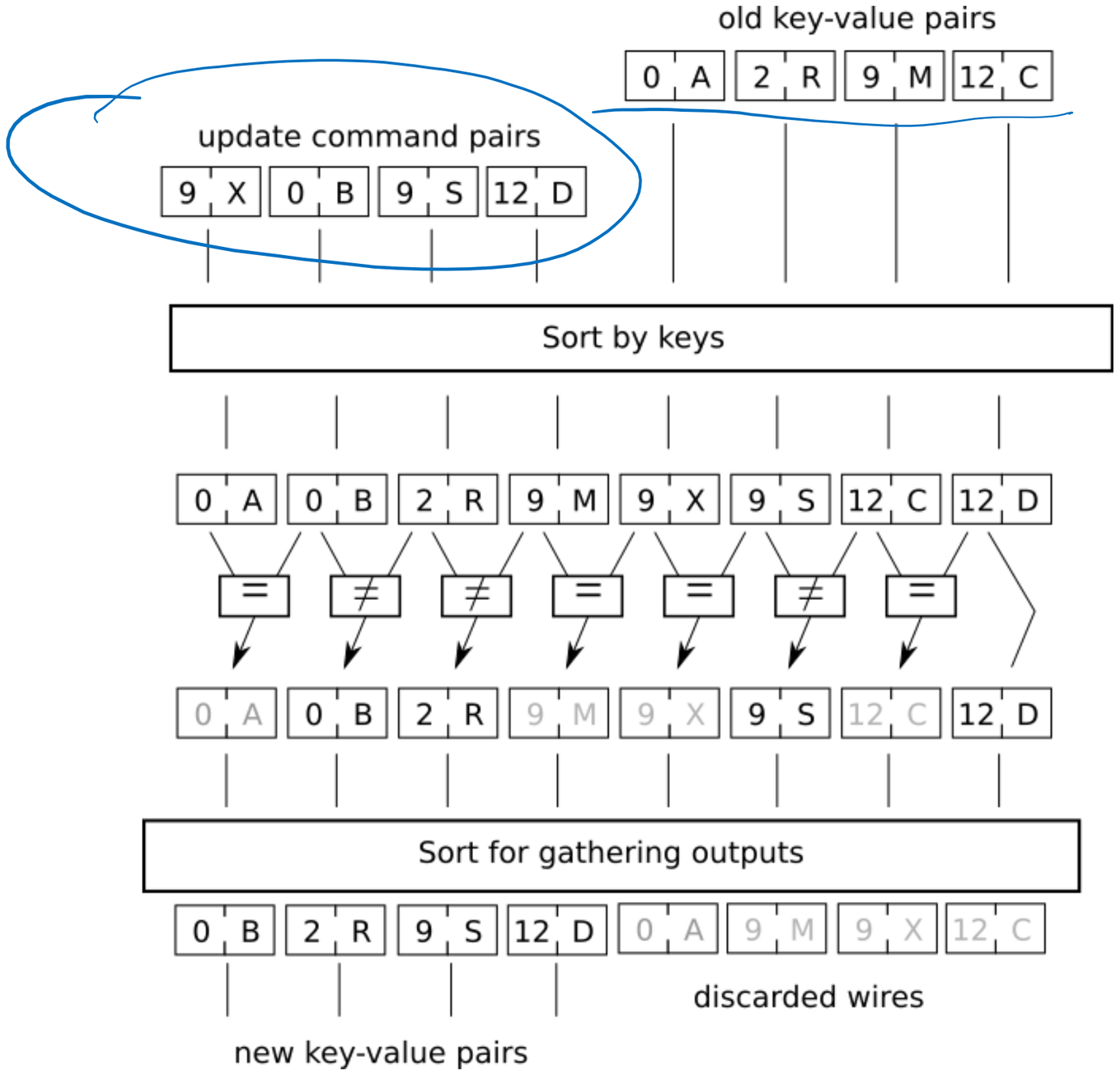
Not just for stacks and queues

Access cost $\Theta(\log n)$



Temporal Batching

$\Theta(n \log^2 n)$



Example Application: **DBScan**

Martin Ester, Hans-Peter Kriegel,
Jörg Sander, Xiaowei Xu. *KDD* 1996

Density-based clustering:

depth-first search to find dense clusters

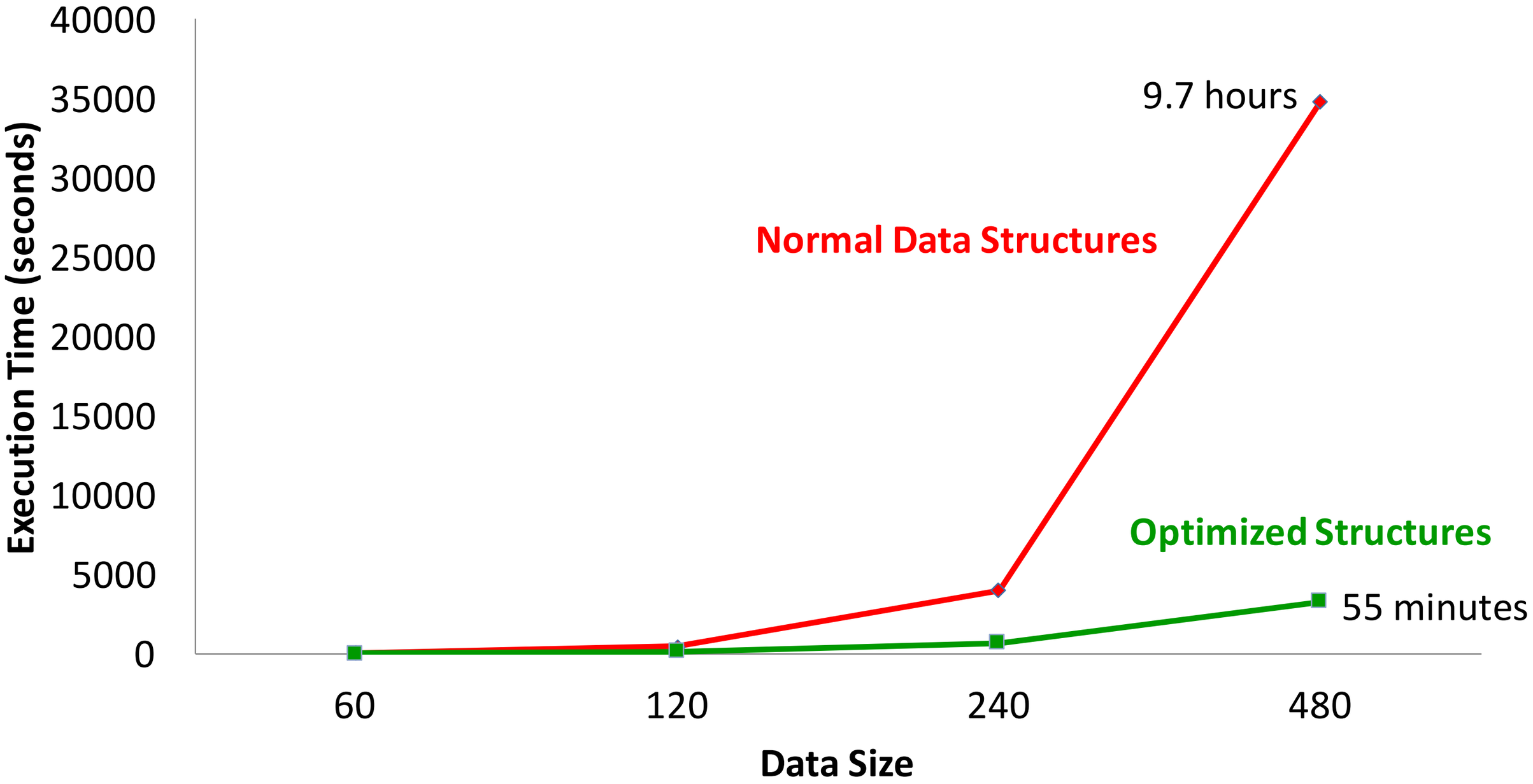
Alice's Data Bob's Data Joint Clusters

```
n ← |P|
c ← 0
s ← emptyStack
cluster ← [0, 0, ...]
for i ← [1, n] do
    if cluster[i] ≠ 0 then
        continue
    V ← getNeighbors(i, P, minpts, radius)
    if count(V) < minpts then
        continue
    c ← c + 1                          ▷ Start a new cluster
    for j ← [1, n] do
        if V[j] = true ∧ cluster[j] ≠ 0 then
            cluster[j] ← c
            s.push(j)
    while s ≠ ∅ do
        k ← s.pop()
        V ← getNeighbors(k, P, minpts, radius)
        if count(V) < minpts then
            continue
        for j ← [1, n] do                          ▷ (C)
            if V[j] = true ∧ cluster[j] ≠ 0 then
                cluster[j] ← c
                s.push(j)
```

Private Input: P – array of points
(combines private points from both parties)
Public inputs: *minpts, radius*
Output: cluster number for each point

Conditional Push!

Array update!


40000
35000
30000
25000
20000
15000
10000
5000
0
Execution Time (seconds)
9.7 hours
Normal Data Structures
Optimized Structures
55 minutes
60
120
240
480
Data Size

Data-Oblivious Memory

Specialized memory access

Circuit structures, protocol agnostic

Stacks, queues, batched map operations

But first…Obliv-C

General random access

Oblivious RAM

Tools for Building Secure Computations

Library-based frameworks:
Circuit-level programs

Full control
Low-level programming
Little type safety

High-level Languages

Little control
High-level programming
Strong type safety

Tools for Building Secure Computations

Library-based frameworks:
Circuit-level programs

Full control
Low-level programming
Little type safety

High-level programming
Low-level customizability
Helpful, **escapable** type checking

High-level Languages

Little control
High-level programming
Strong type safety

Obliv-C

```
#include<obliv.oh>
#include"million.h"

void millionaire(void* args)
{
  protocolIO *io=args;
  obliv int v1,v2;
  bool eq,lt;

  v1 = feedOblivInt(io->mywealth,1);
  v2 = feedOblivInt(io->mywealth,2);
  obliv bool eqx = (v1==v2);
  obliv bool ltx = (v1<v2);
  revealOblivBool(&eq,eqx,0);
  revealOblivBool(&lt,ltx,0);
  io->cmp = (!eq?lt?-1:1:0);
}
```

Obliv-C

```
#include<obliv.oh>
#include"million.h"

void millionaire(void* arg
{
  protocolIO *io=args;
  obliv int v1,v2;
  bool eq,lt;

  v1 = feedOblivInt(io->my
  v2 = feedOblivInt(io->my
  obliv bool eqx = (v1==v2
  obliv bool ltx = (v1<v2)
  revealOblivBool(&eq,eqx,
  revealOblivBool(&lt,ltx,
  io->cmp = (!eq?lt?-1:1:0
}
```

```
#include <million.h>

int main (int argc, char *argv[]) {
    ProtocolDesc pd;
    ProtocolIO io;
    int p = (argv[1] == '1' ? 1 : 2);
    sscanf(argv[2], "%d", &io.myinput);
    // ... set up TCP connections
    setCurrentParty(&pd, p);
    execYaoProtocol(&pd, millionaire, &io);
    printf("Result: %d\n", io->cmp);
    // ... cleanup }
```

Oblivious Conditionals

```
obliv if (cmpresult < 0) {
        iimin = iimid + 1;
} else {
        iimax = iimid;
}
```

from obinary_search

```
obliv int obinary_search_oram(OcCopy * cpy, void* result, oram * haystack, void* needle, block_cmp_function fn) {
        int upper_bound = log2(oram_size(haystack)) + 1;

        obliv int index = -1;
        obliv int iimin = 0;
        obliv int iimax = oram_size(haystack) - 1;
        obliv int iimid;
        obliv int cmpresult;
        void * temp_element = calloc(1, cpy->eltsize);
        for (int ii = 0; ii < upper_bound; ii++) {
                iimid = (iimin + iimax) / 2;
                oram_read(temp_element, haystack, iimid);
                cmpresult = fn(cpy, temp_element, needle);
                obliv if (cmpresult == 0) {
                        ocCopy(cpy, result, temp_element);
                        index = iimid;
                } else {
                        obliv if (cmpresult < 0) {
                                iimin = iimid + 1;
                        } else {
                                iimax = iimid;
                        }
                }
        }
        free(temp_element);
        return index;
}
```

Actual code…with all the ugly parts

Escaping

```
~obliv(var) {
        …
}
```

Code inside ~obliv always executes regardless of oblivious condition

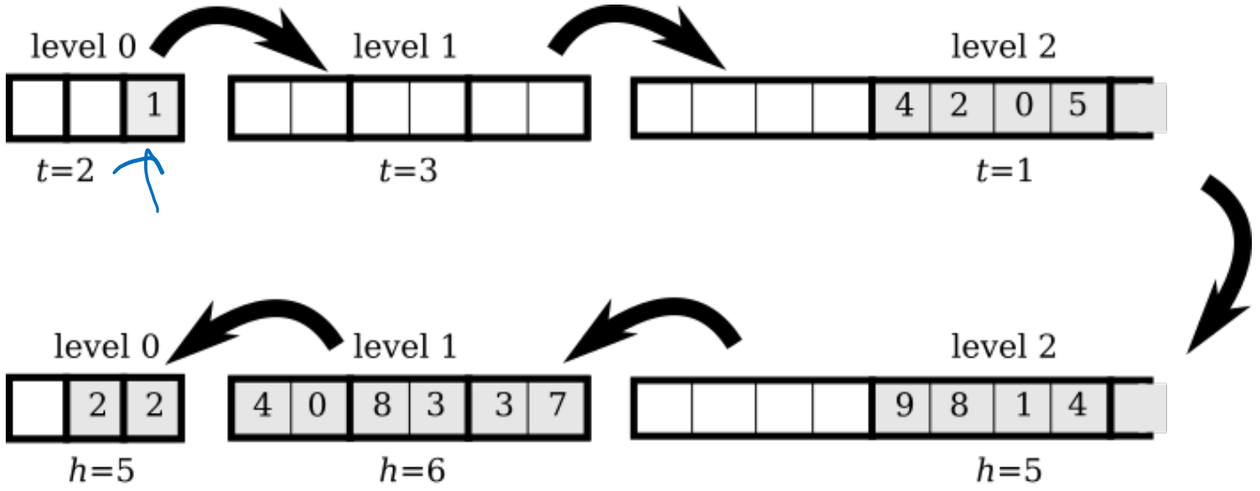
var is Boolean: oblivious condition

Programmer has control! But, not security risk: all private data is still encrypted

Implementing Oblivious Queue

```
#include <math.h>
#include "oqueue.oh"

struct oqueue {
        oqueue * child;
        OcCopy * cpy;
        size_t layer_index;
        size_t copymult;
        bool dynamic_sizing;
        obliv uint8_t head;
        obliv uint8_t tail;
        obliv uint32_t current_elements;
        uint8_t push_time;
        uint8_t pop_time;
        void * data;
};
```



```
obliv bool oqueue_push(oqueue * layer, void * input) obliv {
        obliv bool success = false;
        ~obliv(en) {
                if (layer->child != NULL) {
                        if (layer->push_time == 1) {
                                obliv if (layer->tail >= 5) {
                                        obliv if (oqueue_push(layer->child, element(layer->cpy,layer->data,3*layer->copymult))) {
                                                ocCopyN(layer->cpy,element(layer->cpy,layer->data,3*layer->copymult), element(layer-
                                                layer->tail -= 2;
                                        }
                                }
                                layer->push_time = 0;
                        } else {
                                layer->push_time ++;
                        }
                }
        }
        obliv if (layer->tail < 6) {
                for (uint8_t ii = 0; ii < 6; ii++) {
                        obliv if (ii == layer->tail) {
                                ocCopyN(layer->cpy,element(layer->cpy,layer->data,ii * layer->copymult), input, layer->copymult);
                                success = true;
                                layer->current_elements ++;
                        }
                }
                layer->tail++;
        }
        return success;
}
```

Obliv-C

http://oblivc.org/

Historical Excursion

A Permutation Network

ABRAHAM WAKSMAN

Stanford Research Institute, Menlo Park, California

ABSTRACT. In this paper the construction of a switching network capable of $n!$-permutation of its n input terminals to its n output terminals is described. The building blocks for this network are binary cells capable of permuting their two input terminals to their two output terminals.

The number of cells used by the network is $\langle n \cdot \log_2 n - n + 1 \rangle = \sum_{k=1}^{n} \langle \log_2 k \rangle$. It could be argued that for such a network this number of cells is a lower bound, by noting that binary decision trees in the network can resolve individual terminal assignments only and not the partitioning of the permutation set itself which requires only $\langle \log_2 n! \rangle = \langle \sum_{k=1}^{n} \log_2 k \rangle$ binary ...ons.

Journal of the ACM, January 1968

1967 ACM Turing Lecture

Computers Then and Now

MAURICE V. WILKES

Cambridge University, Cambridge, England

(In same Jan 1968 JACM as Waksman Network!)

	READ	WRITE	RECIRCULATE
READ SELECT	1	0 OR 1	0 OR 1
WRITE SELECT	0 OR 1	1	0
DATA INPUT	0 OR 1	0 OR 1	0 OR 1
R/W ENABLE	1	1	0 OR 1

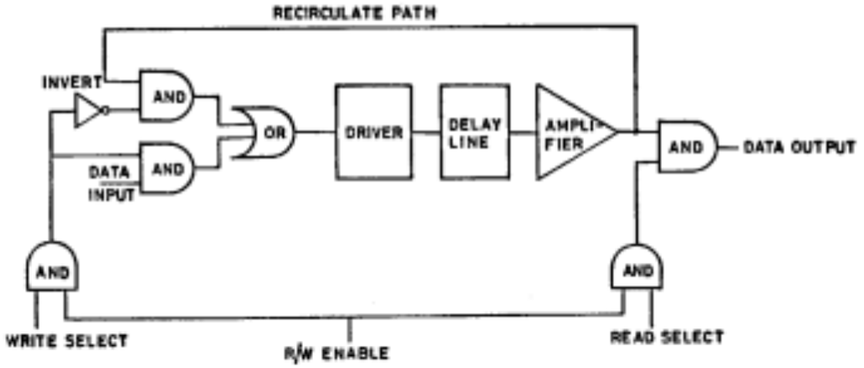
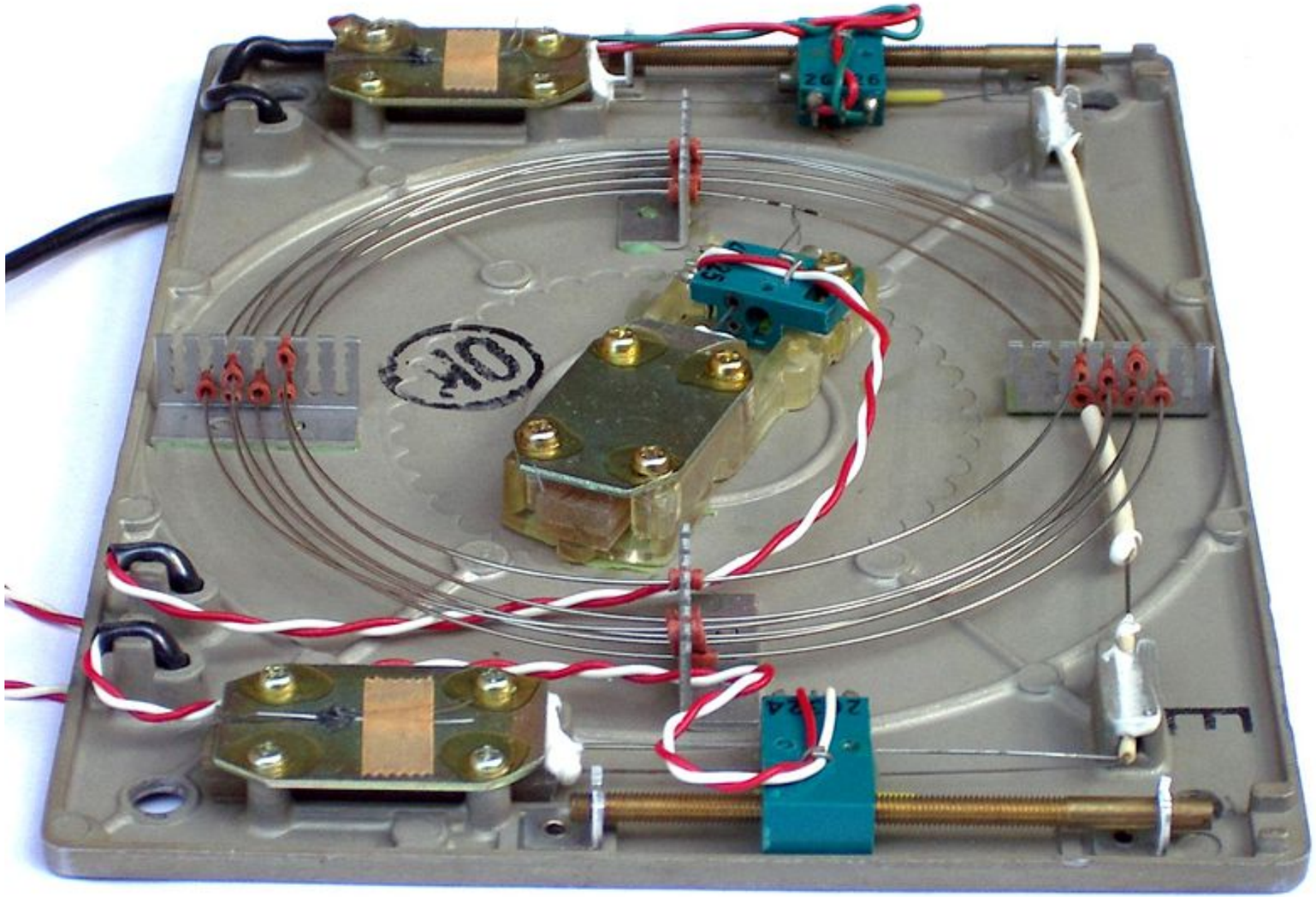
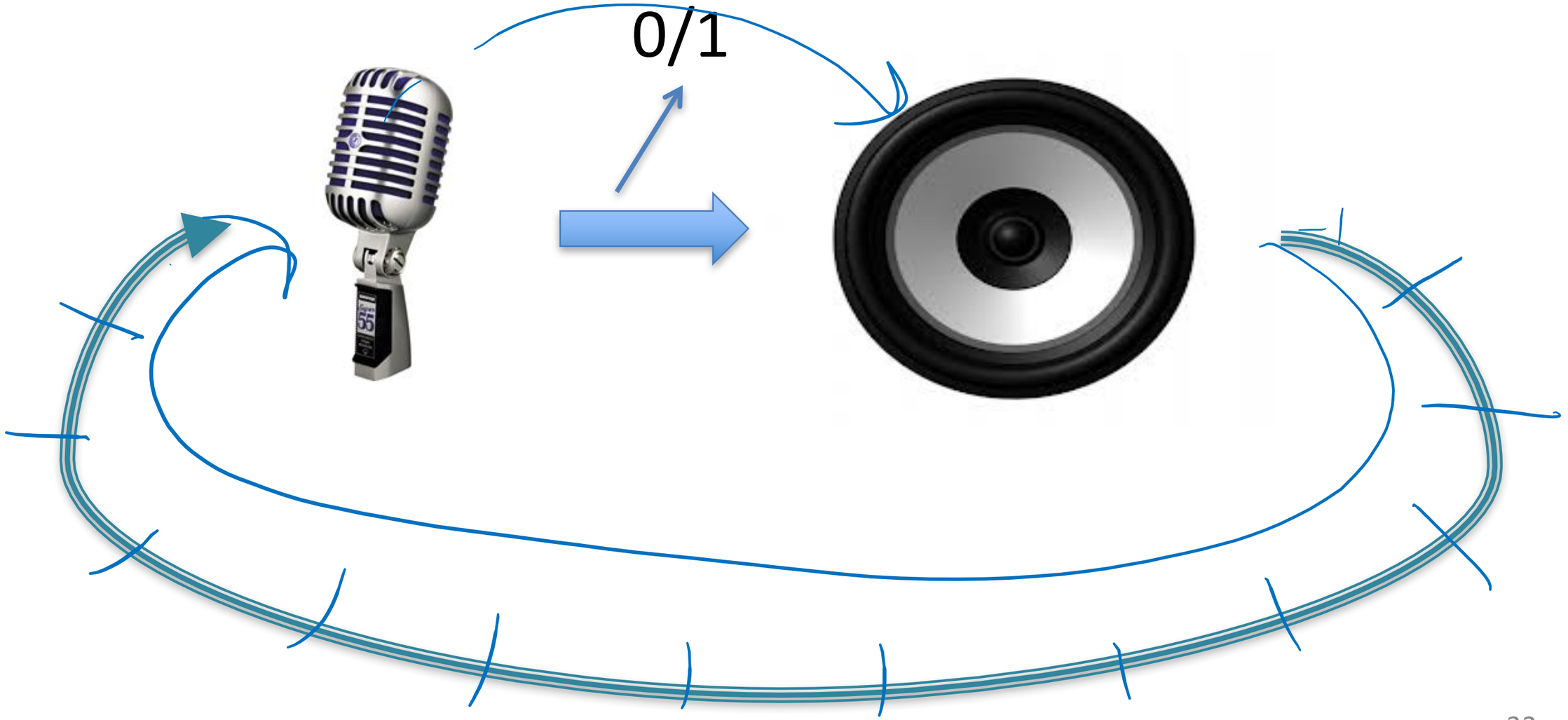


FIG. 2.1.1. Delay line memory block diagram.

Delay Lines



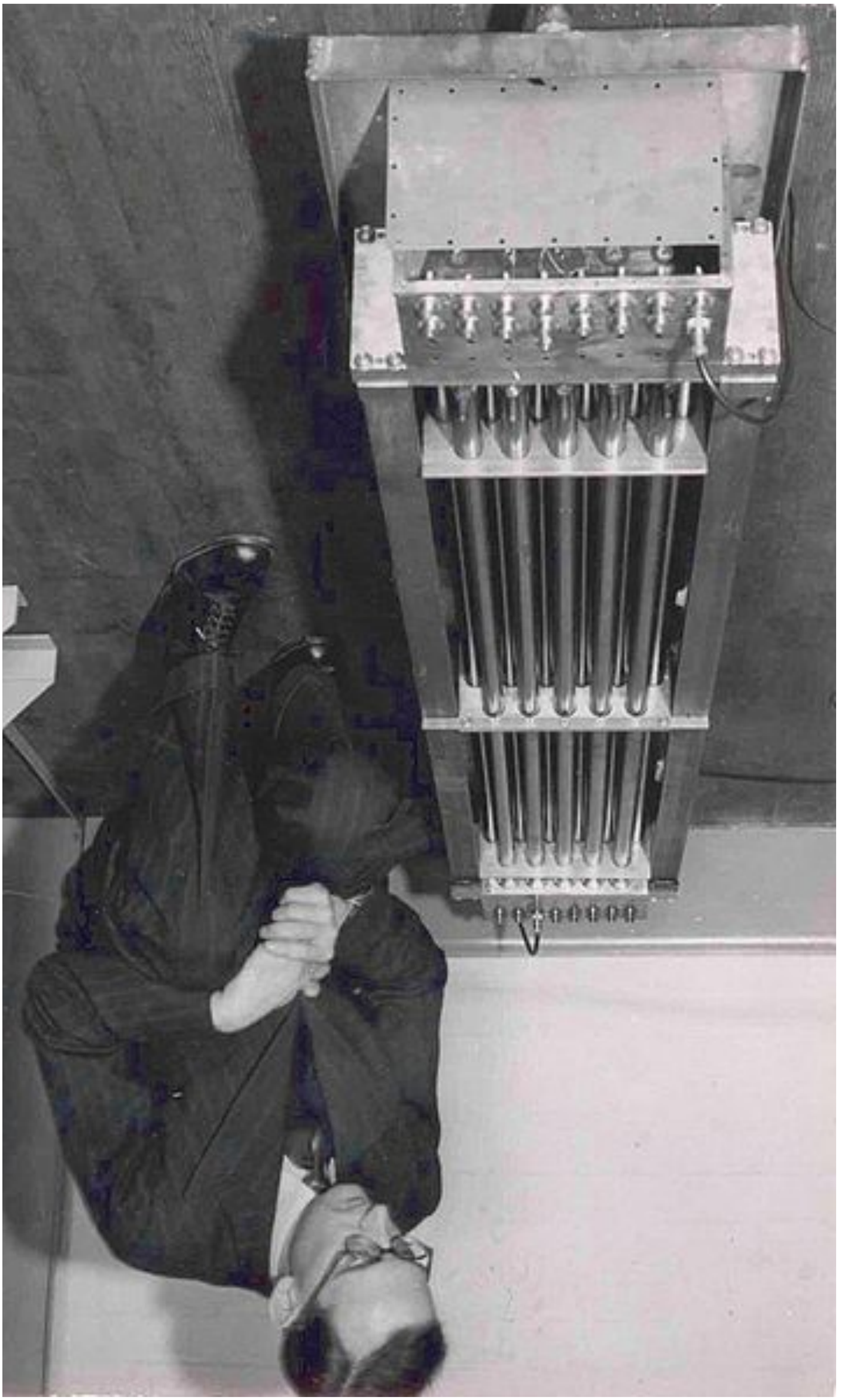
Mercury Delay Lines



Speed of Sound	
Air	343 m/s
Mercury	1450 m/s (40° C)
Water	1500 m/s (25° C)

Why Mercury?

Turing's contribution to this discussion was to advocate the use of gin, which he said contained alcohol and water in just the right proportions to give a zero temperature coefficient of propagation velocity at room temperature.



and tribulations that we are now going through.

In ultrasonic memories, it was customary to store up to 32 words end to end in the same delay line. The pulse rate was fairly high, but people were much worried about the time spent in waiting for the right word to come around. Most delay line computers were, therefore, designed so that, with the exercise of cunning, the programmer could place his instructions and numbers in the memory in such a way that the waiting time was minimized. Turing himself was a pioneer in this type of logical design. Similar methods were later applied to computers which used a magnetic drum as their memory and, altogether, the subject of *optimum coding*, as it was called, was a flourishing one. I felt that this kind of human ingenuity was misplaced as a long-term investment, since sooner or later we would have truly random-access memories. We therefore did not have anything to do with optimum coding in Cambridge.

Magnetic Core Memory

MIT Project Whirlwind, 1951
2K 16-bit words
with “**no waiting**”!

Oblivious RAM

called, was a flourishing one. I felt that this kind of human ingenuity was misplaced as a long-term investment, since sooner or later we would have truly random-access memories. We therefore did not have anything to do with optimum coding in Cambridge.

Linear Scan Doesn't Scale

Writing a single 32-bit integer: 32 logic gates

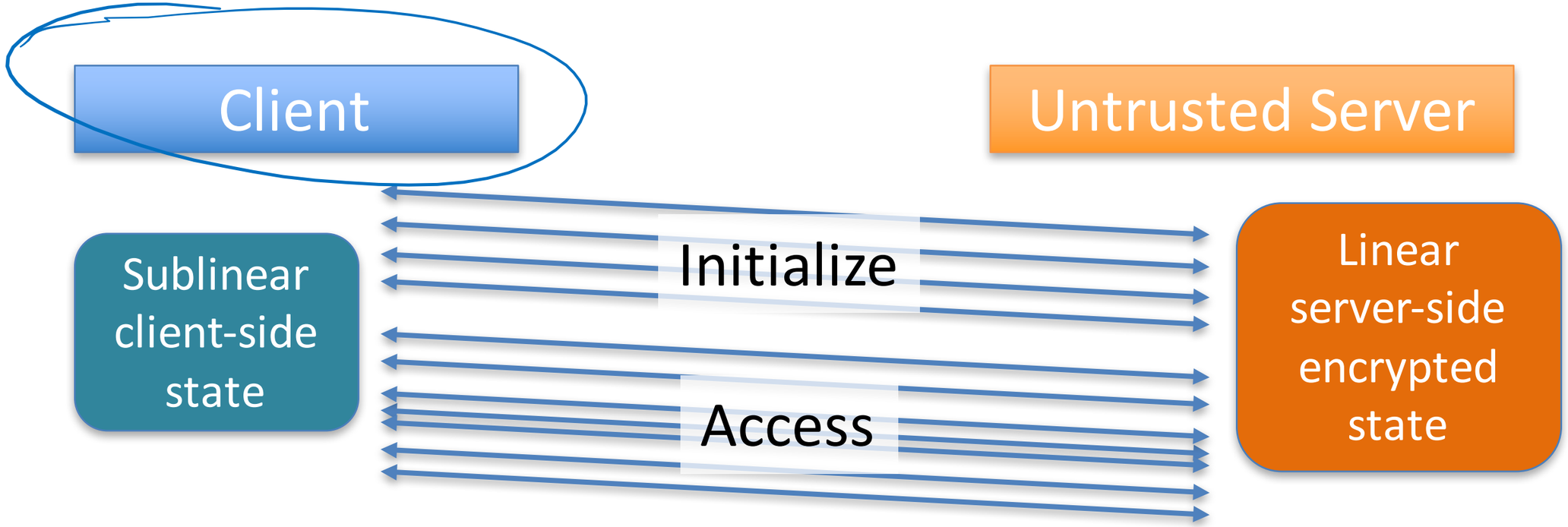
Raw Yao's performance ≈ 3M gates per second

Write speed ≈ 100,000 elements per second

(not hiding access pattern)

For hiding access pattern, N = 2^{17} elements requires > 1 **second per access**

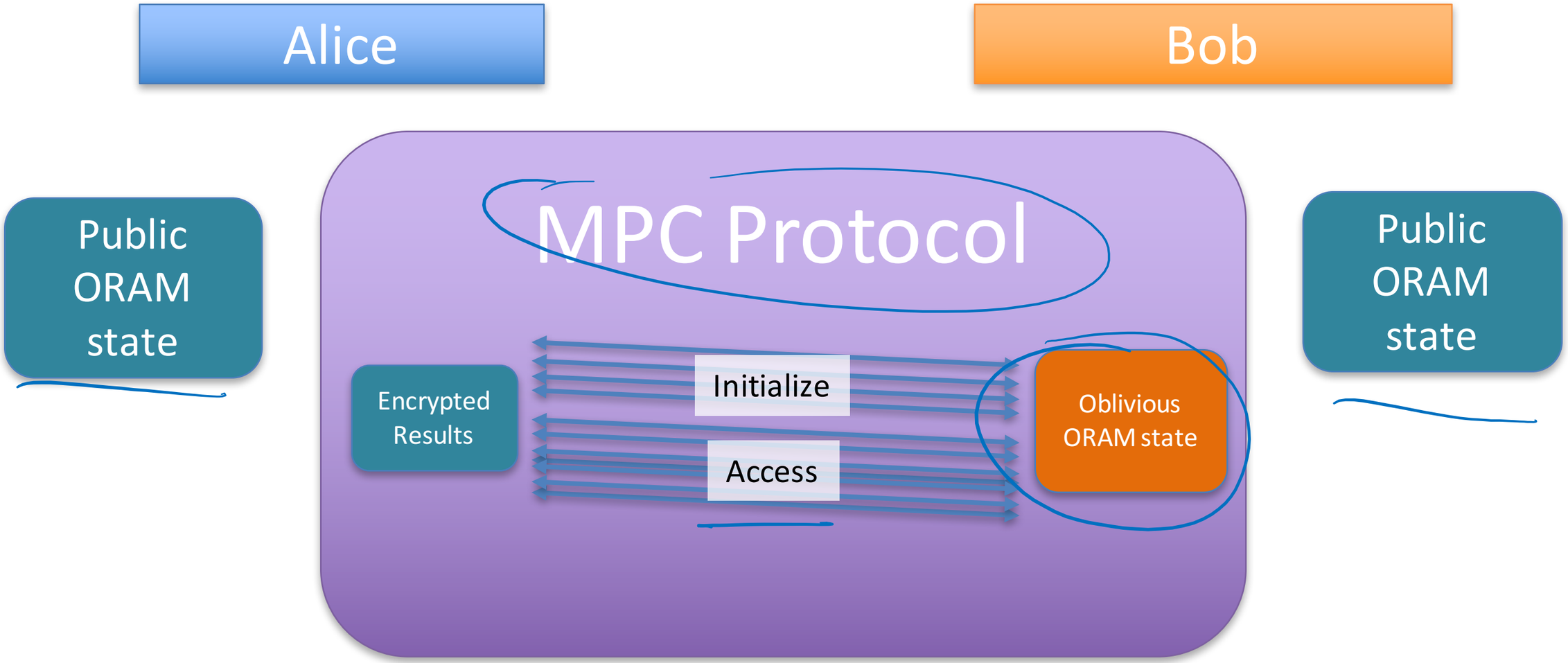
Traditional ORAM [Goldreich 1987]



Security property: all initialization and access sequences of the same length are indistinguishable to server.

RAM-SC

[Gordon, Katz, Kolesnikov, Krell, Malkin, Raykova, Vahlis 2012]

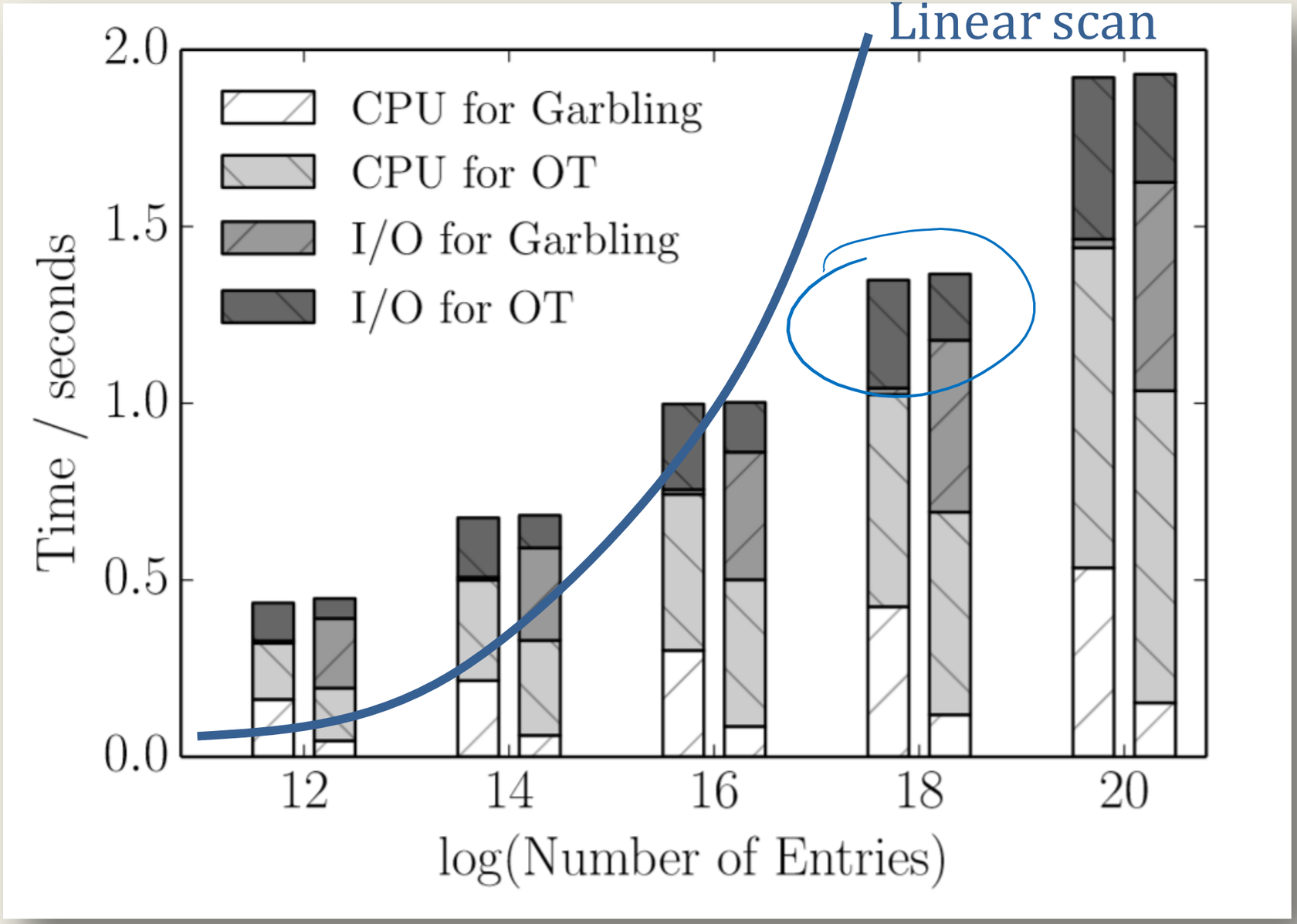


Circuit ORAM Access time

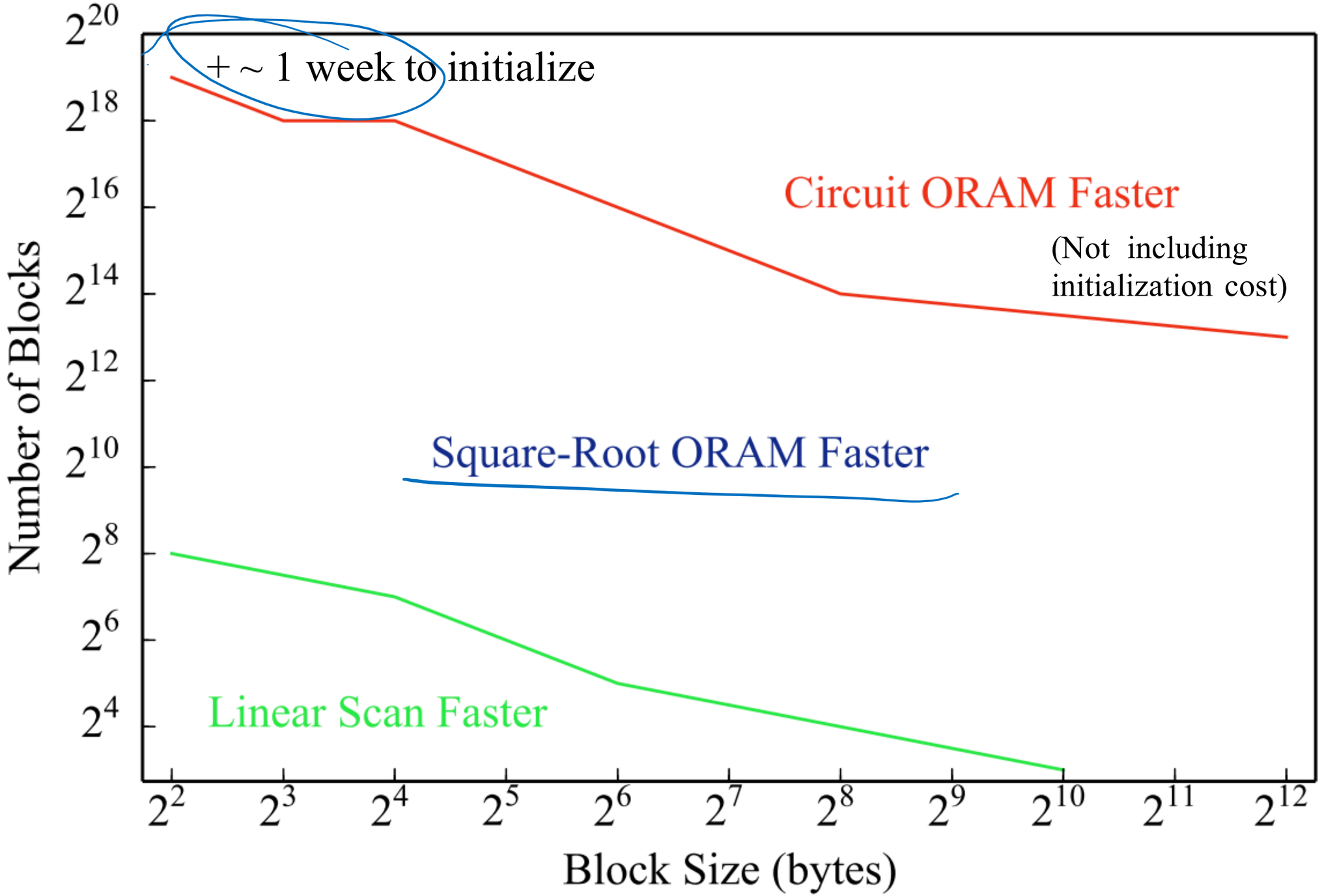
State-of-the-ORAM-Art in 2015

Xiao Wang, Hubert Chan, and Elaine Shi. *Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound*. In ACM CCS 2015.

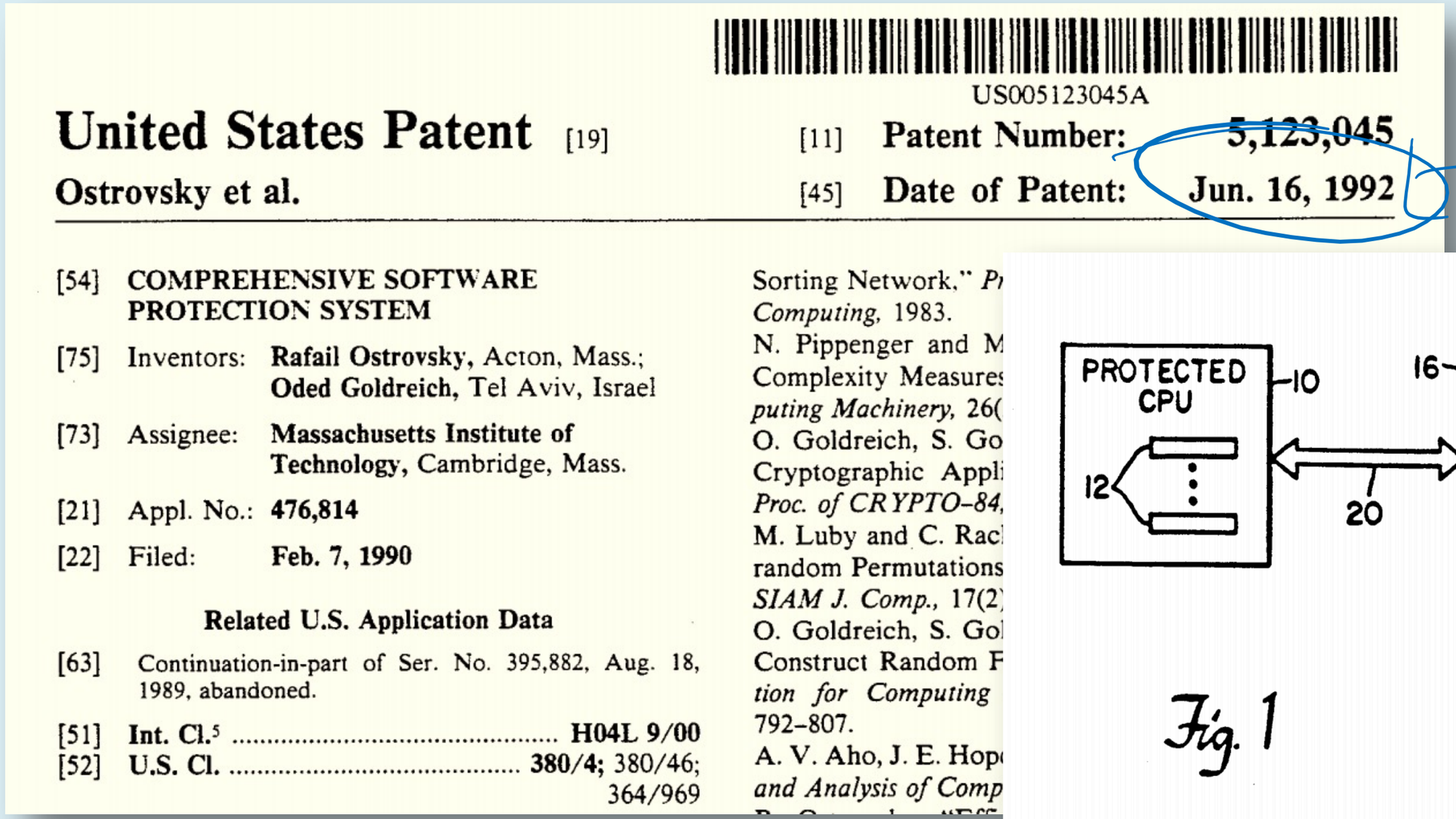
$\Theta(\log^3 n)$



Results Summary



Classical Square-Root ORAM



US005123045A

United States Patent [19]

Ostrovsky et al.

[11] **Patent Number: 5,123,045**

[45] **Date of Patent: Jun. 16, 1992**

Expired 2012

[54] **COMPREHENSIVE SOFTWARE PROTECTION SYSTEM**

[75] Inventors: **Rafail Ostrovsky**, Acton, Mass.; **Oded Goldreich**, Tel Aviv, Israel

[73] Assignee: **Massachusetts Institute of Technology**, Cambridge, Mass.

[21] Appl. No.: **476,814**

[22] Filed: **Feb. 7, 1990**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 395,882, Aug. 18, 1989, abandoned.

[51] **Int. Cl.**[5] **H04L 9/00**

[52] **U.S. Cl.** **380/4**; 380/46; 364/969

Sorting Network," *Pr*
Computing, 1983.
N. Pippenger and M
Complexity Measures
puting Machinery, 26(
O. Goldreich, S. Go
Cryptographic Appli
Proc. of CRYPTO-84,
M. Luby and C. Rac
random Permutations
SIAM J. Comp., 17(2)
O. Goldreich, S. Go
Construct Random F
tion for Computing
792–807.
A. V. Aho, J. E. Hop
and Analysis of Comp

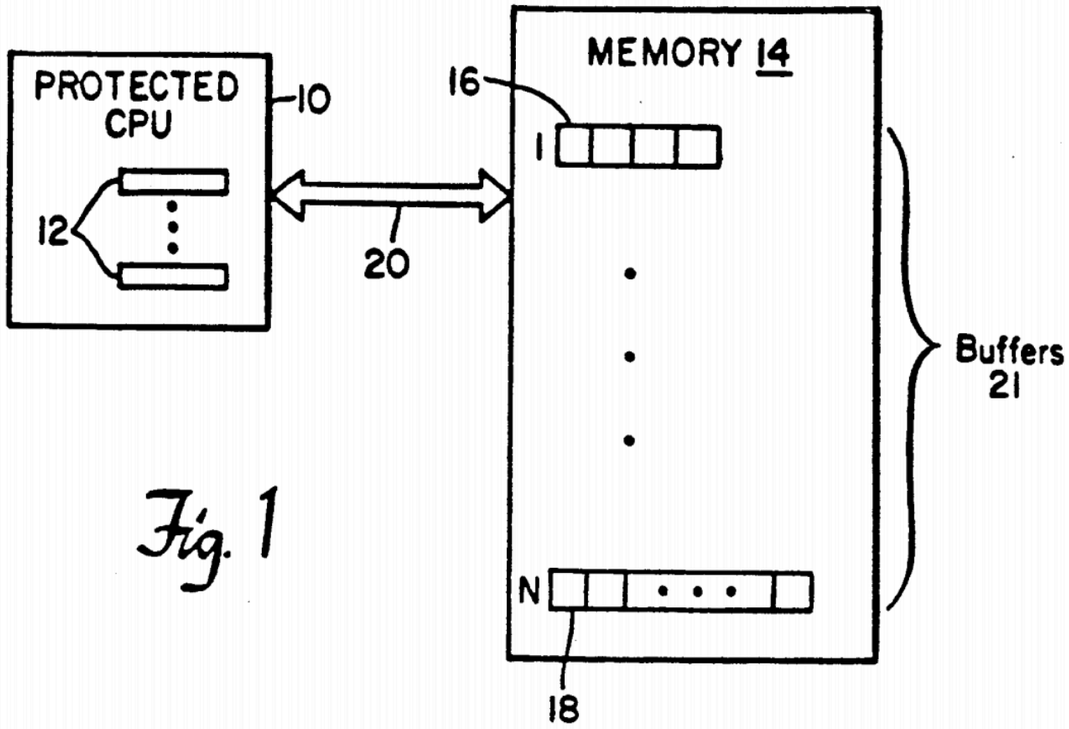


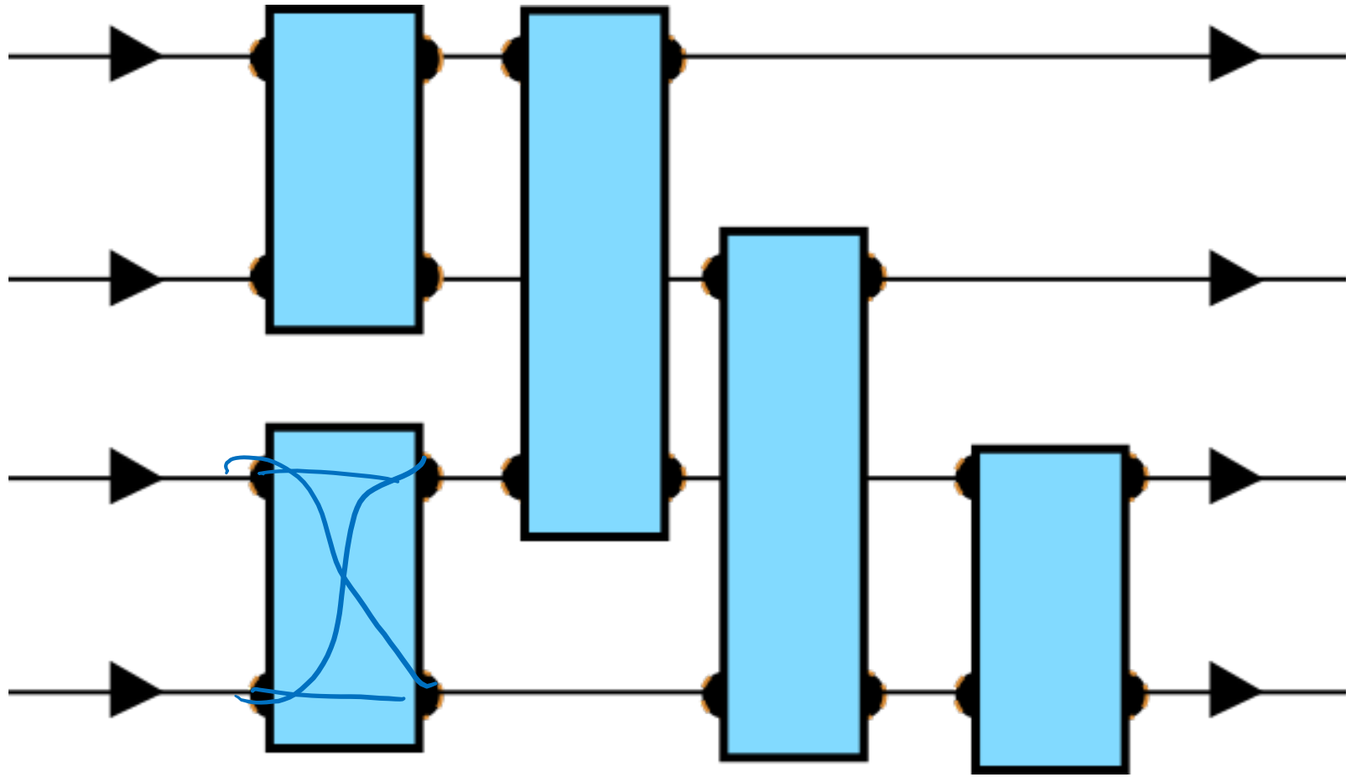
Fig. 1

Problems with SQ-ORAM Design

- Requires a PRF for each ORAM access
  - PRF is a big circuit in MPC
- Initialization requires $N + \sqrt{N}$ PRF evaluations
- Requires $\Theta(Nlog^2N)$ oblivious sort twice:
  - Shuffling memory according to PRF
  - Removing dummy blocks

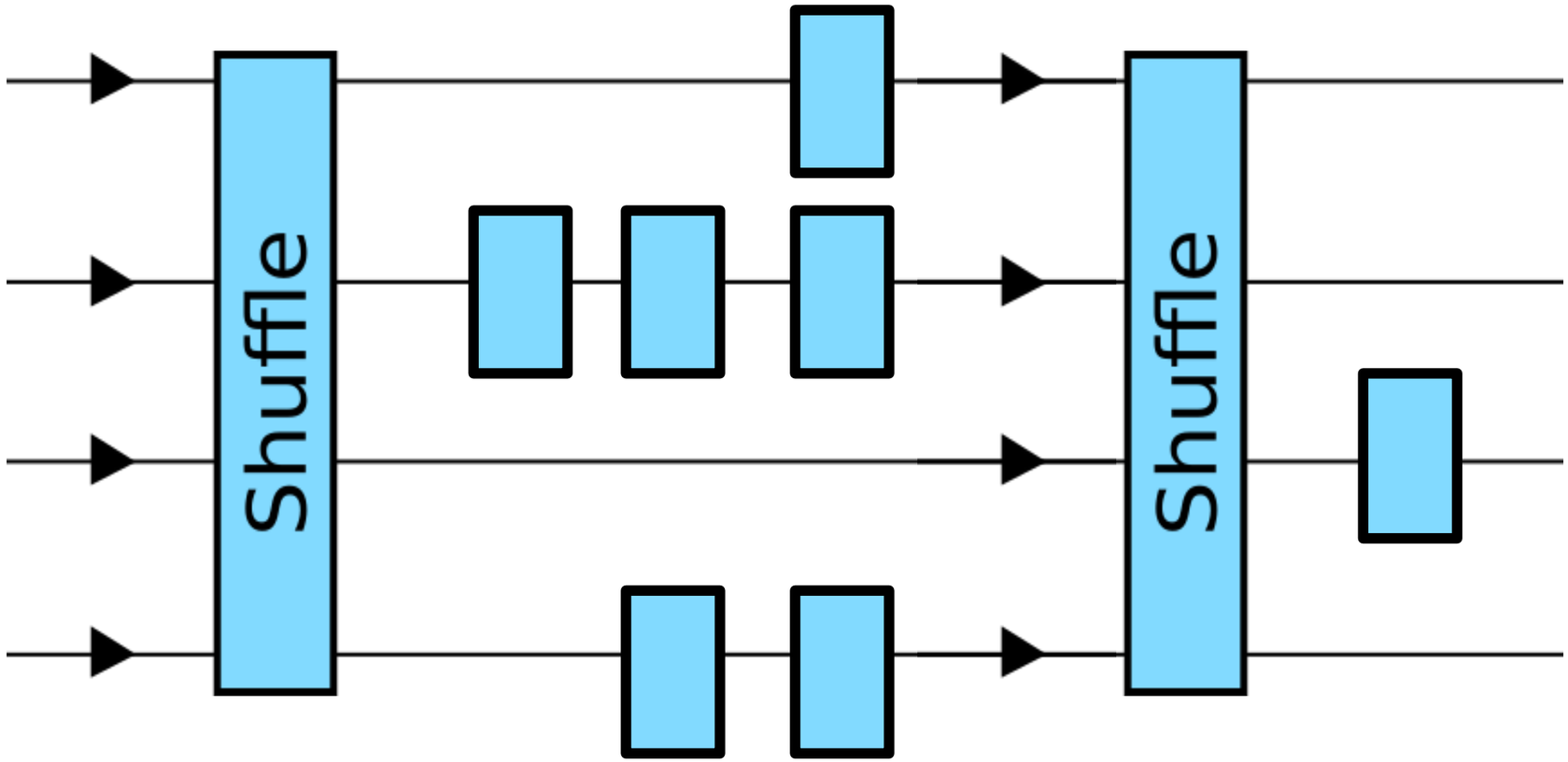
Solution strategy: use random permutation instead of PRF

Shuffling Network [Waksman 1968]

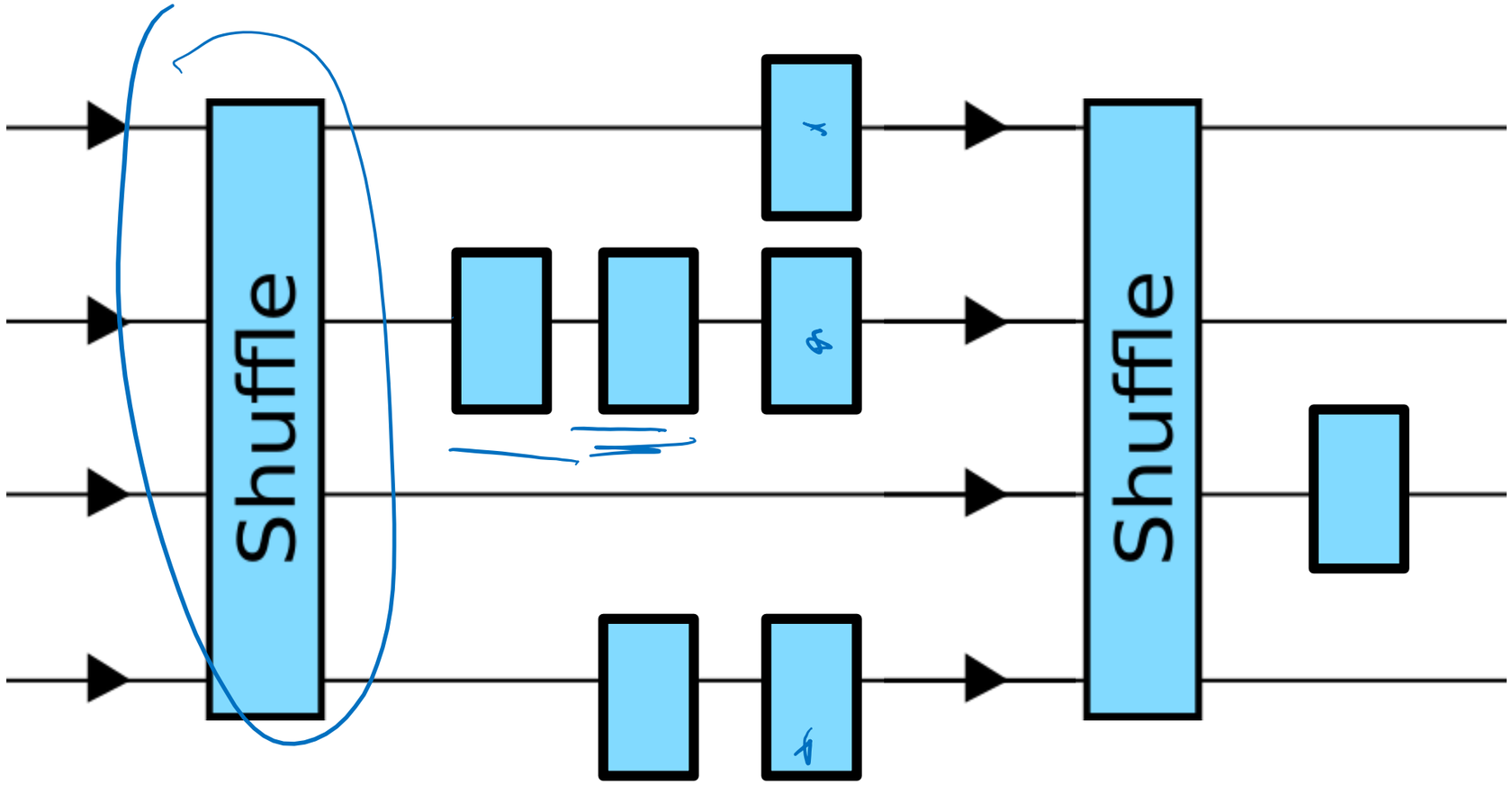


Cost per shuffle: $5B$

4-Block ORAM



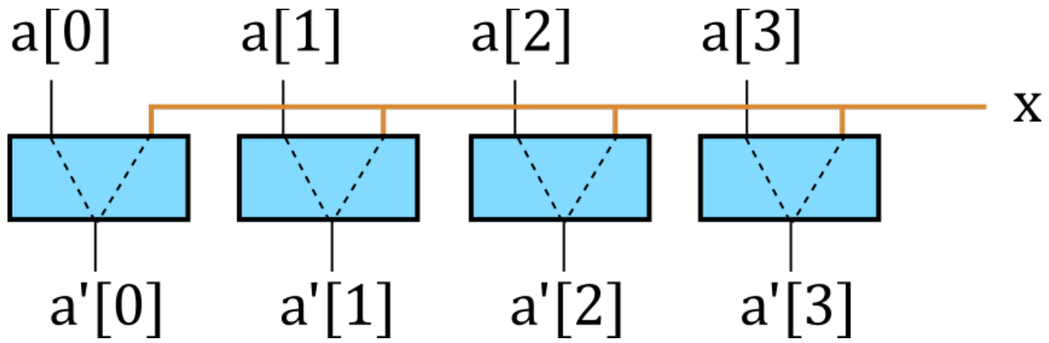
4-Block ORAM



Cost: $5B$ + B $+2B$ $+3B$ + …

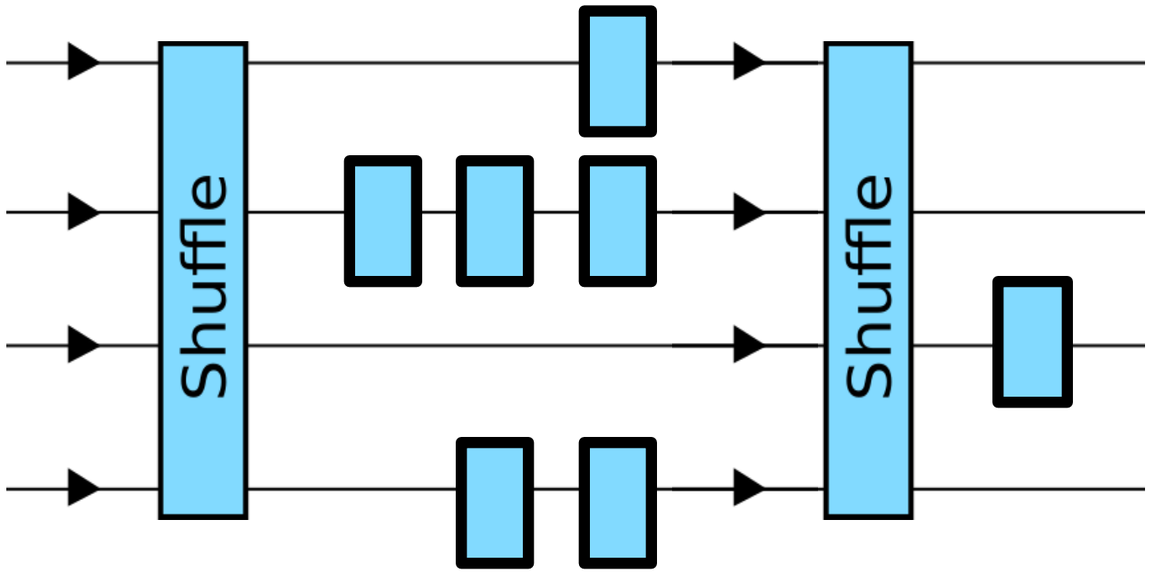
= $11B$ every 3 accesses

Linear scan



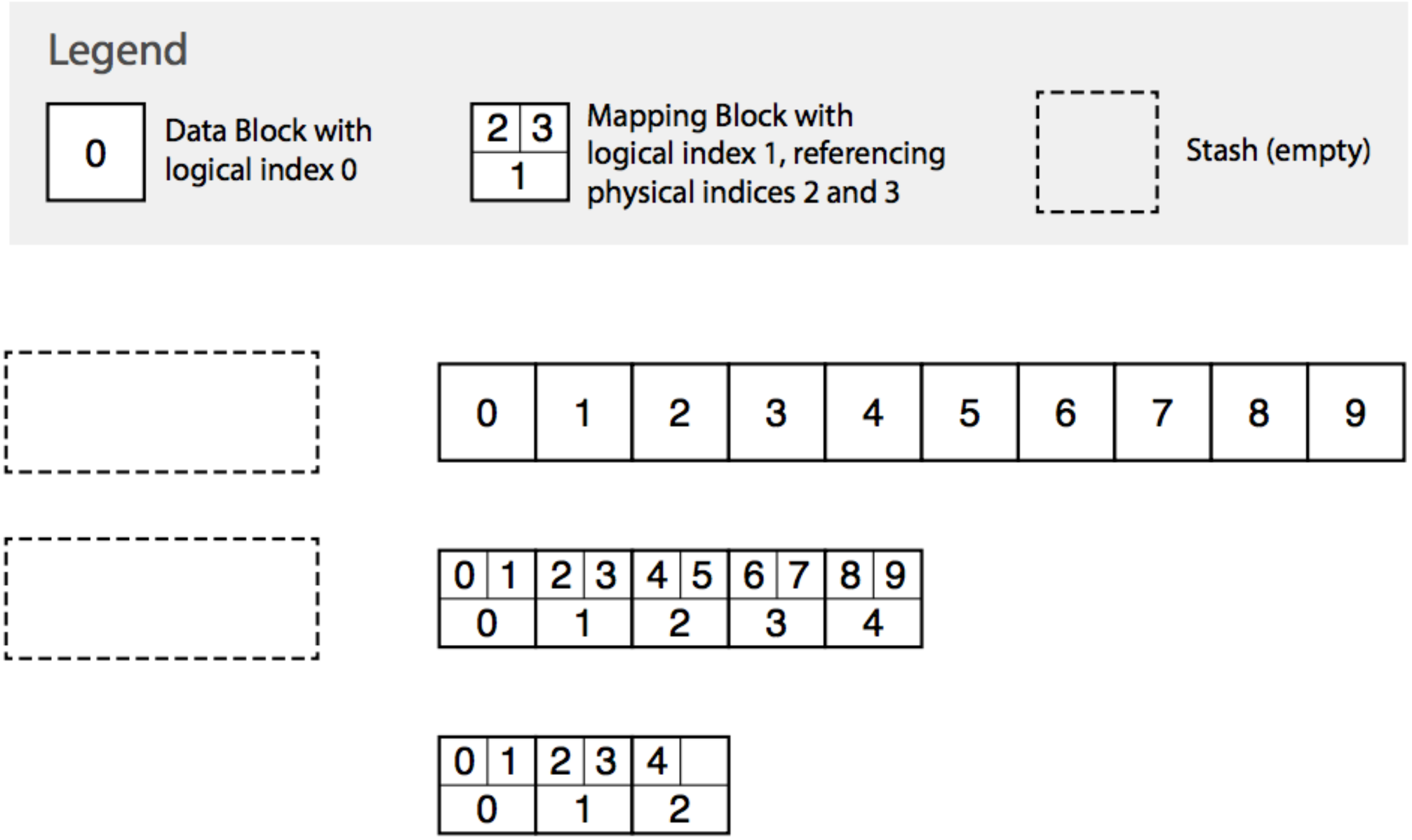
Cost: $4B = 12B/3$

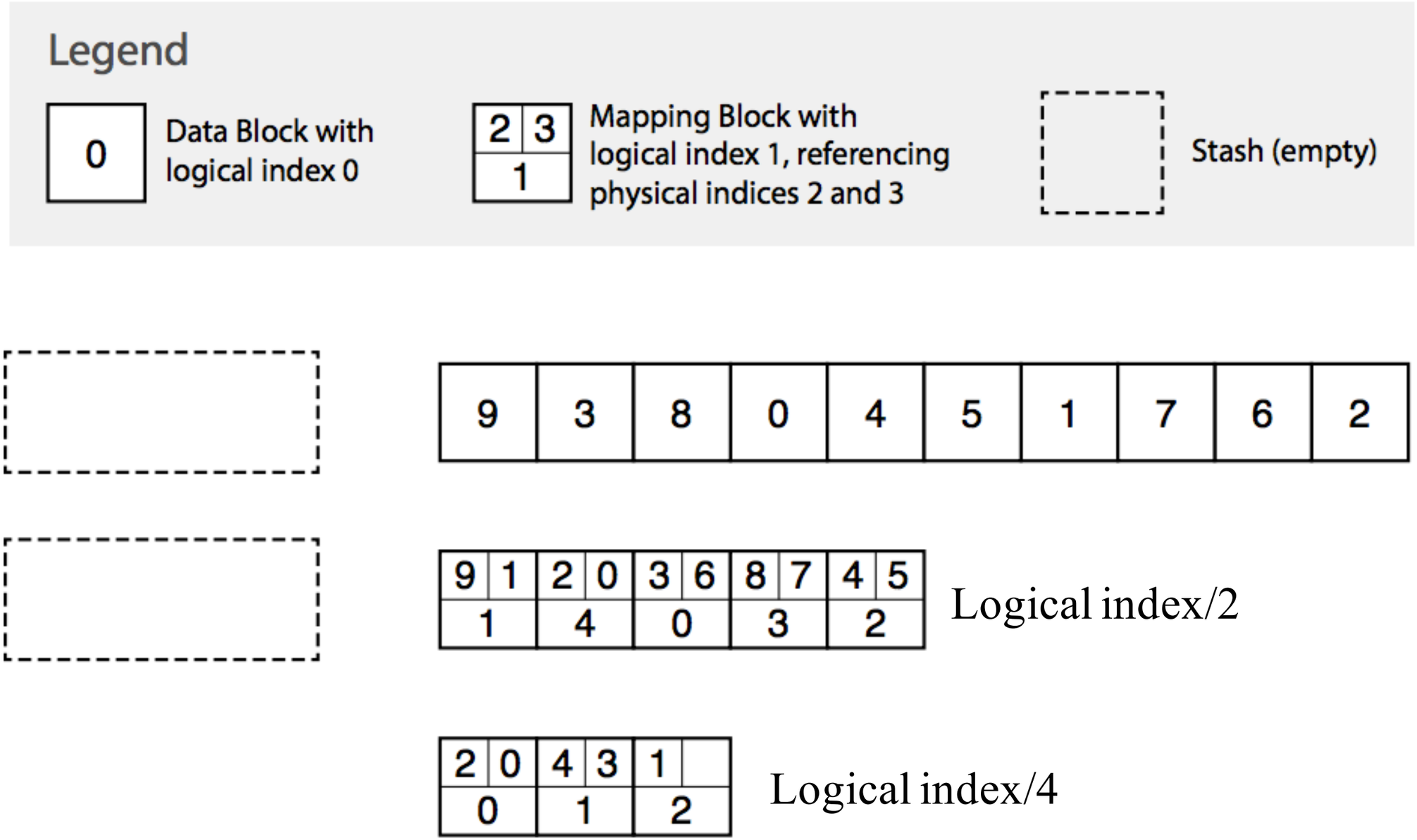
Our scheme



Cost: $11B/3$

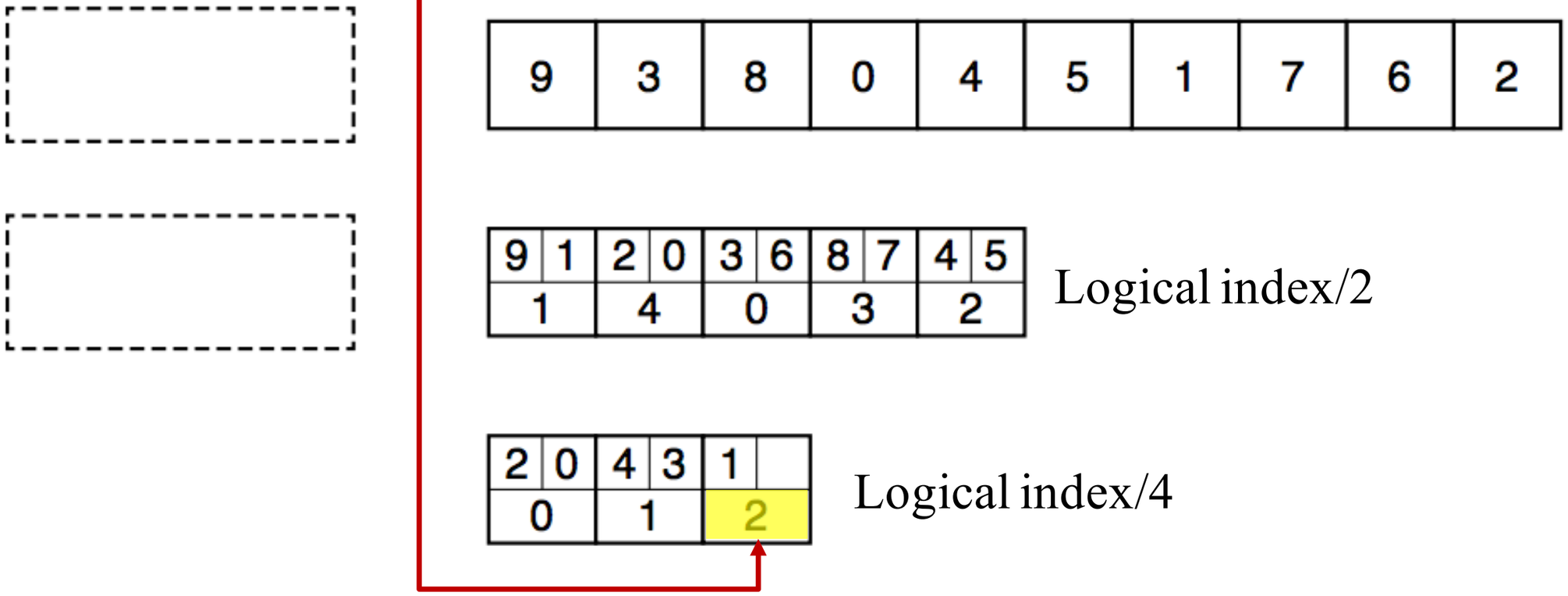
Less expensive than linear scan for 4 blocks (8 with overhead)


Legend
0
Data Block with logical index 0
2 3
1
Mapping Block with logical index 1, referencing physical indices 2 and 3
Stash (empty)
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2

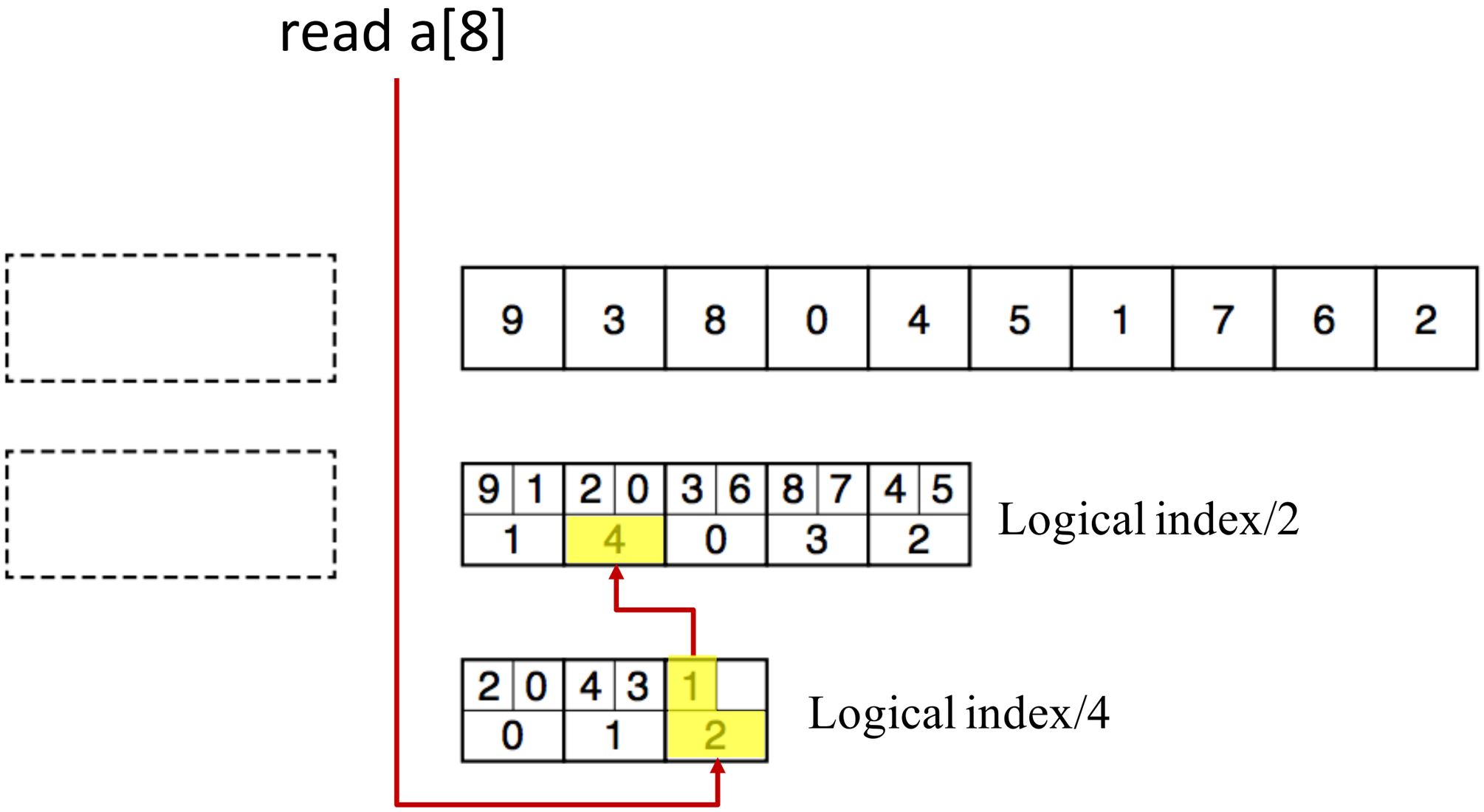

Legend
0
Data Block with logical index 0
2 3
1
Mapping Block with logical index 1, referencing physical indices 2 and 3
Stash (empty)
9 3 8 0 4 5 1 7 6 2
9 1 2 0 3 6 8 7 4 5
1 4 0 3 2
Logical index/2
2 0 4 3 1
0 1 2
Logical index/4

First Access

read a[8]



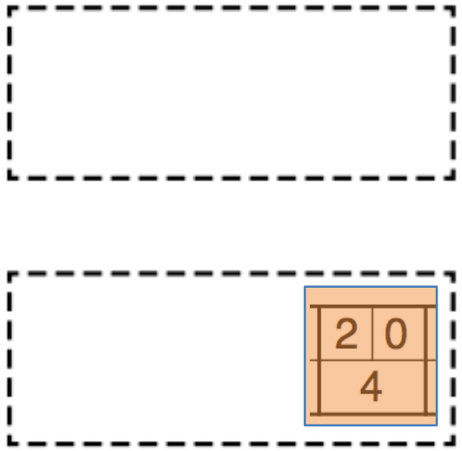
First Access

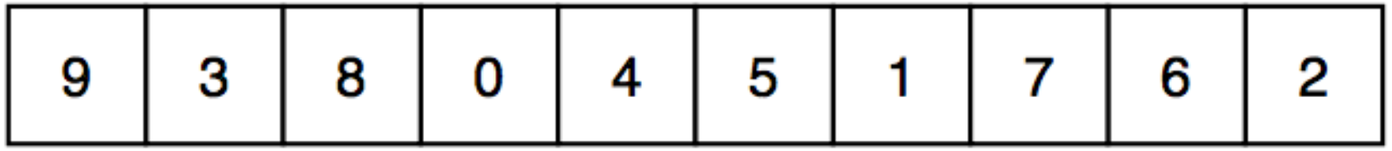


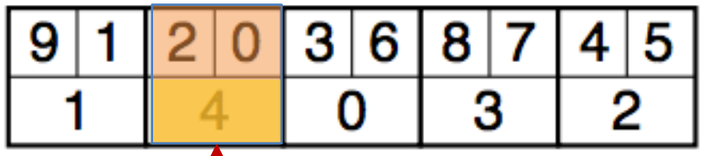
First Access

read a[8]

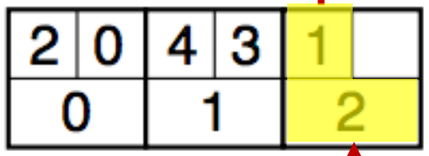
stash





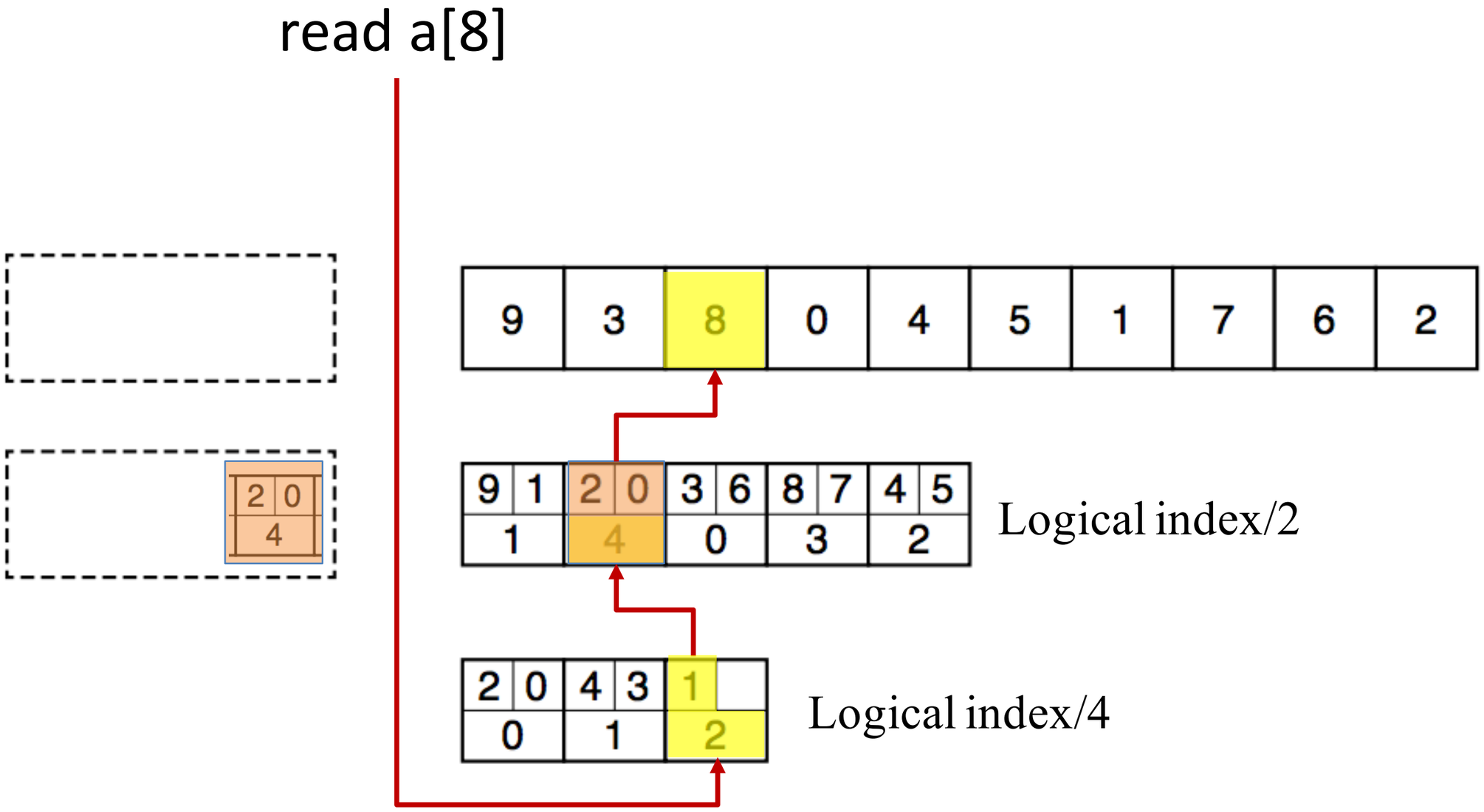


Logical index/2

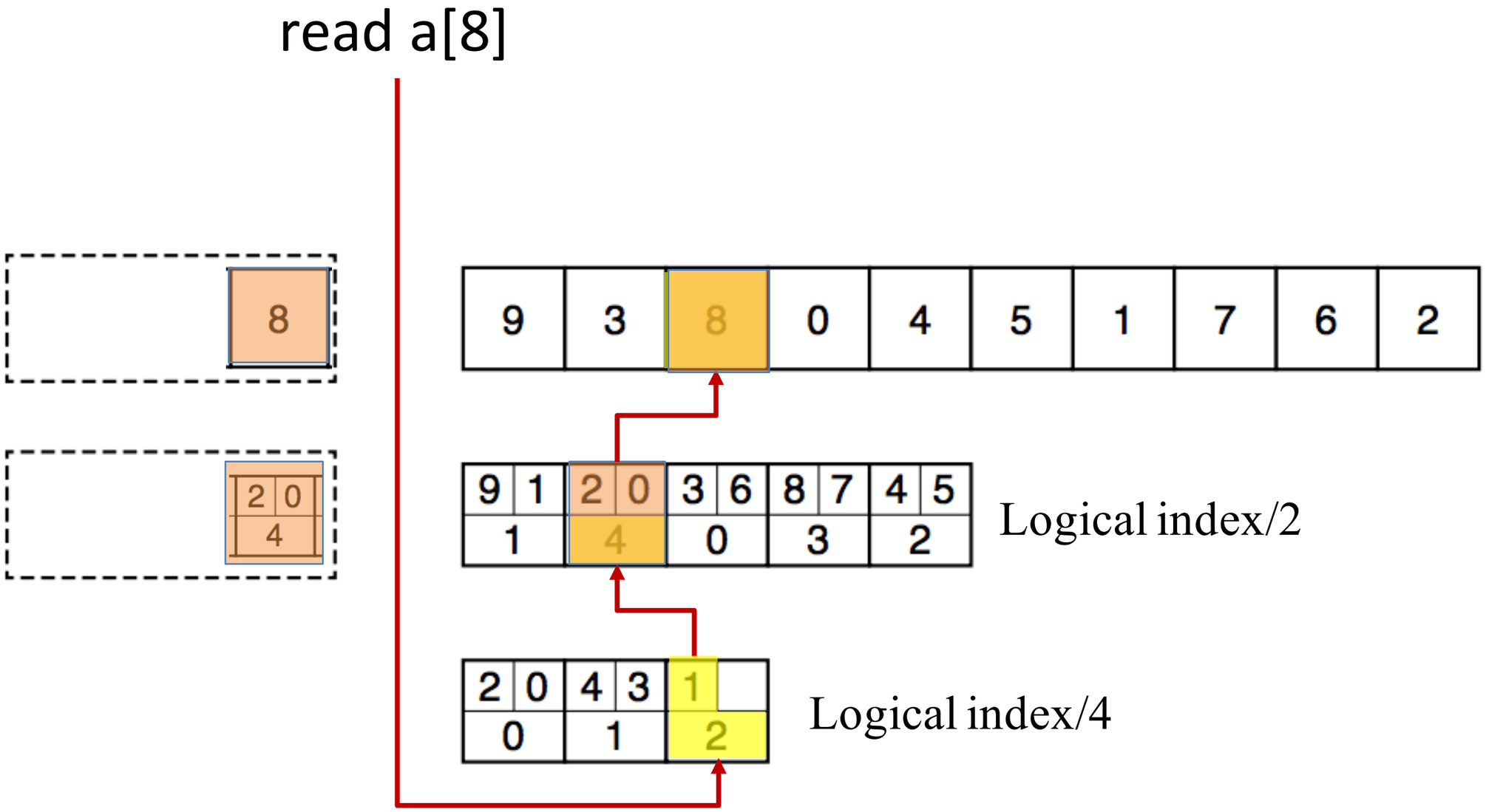


Logical index/4

First Access

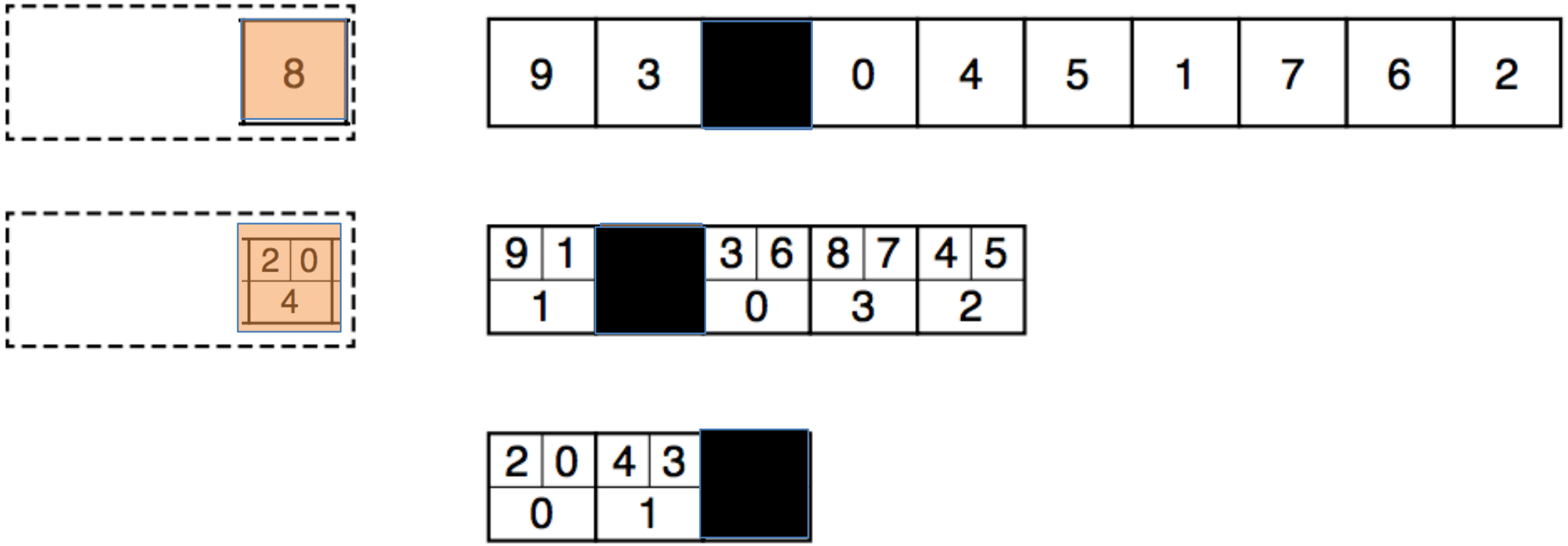


First Access



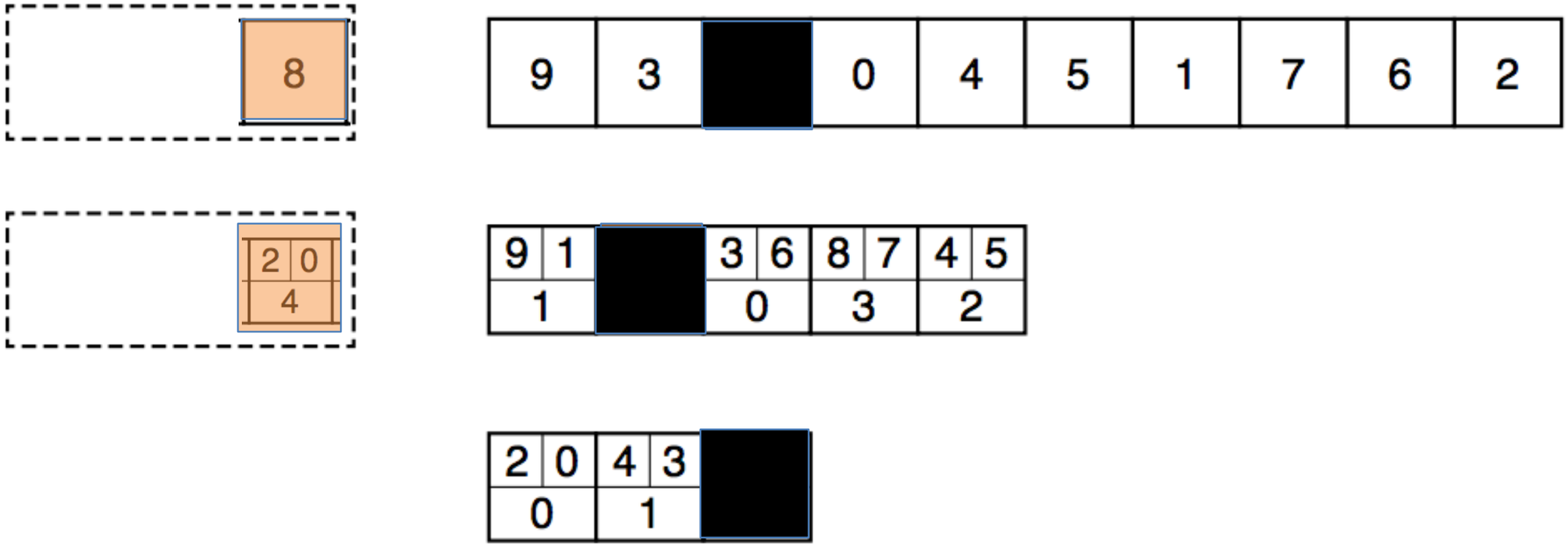
After First Access

Used (Public)

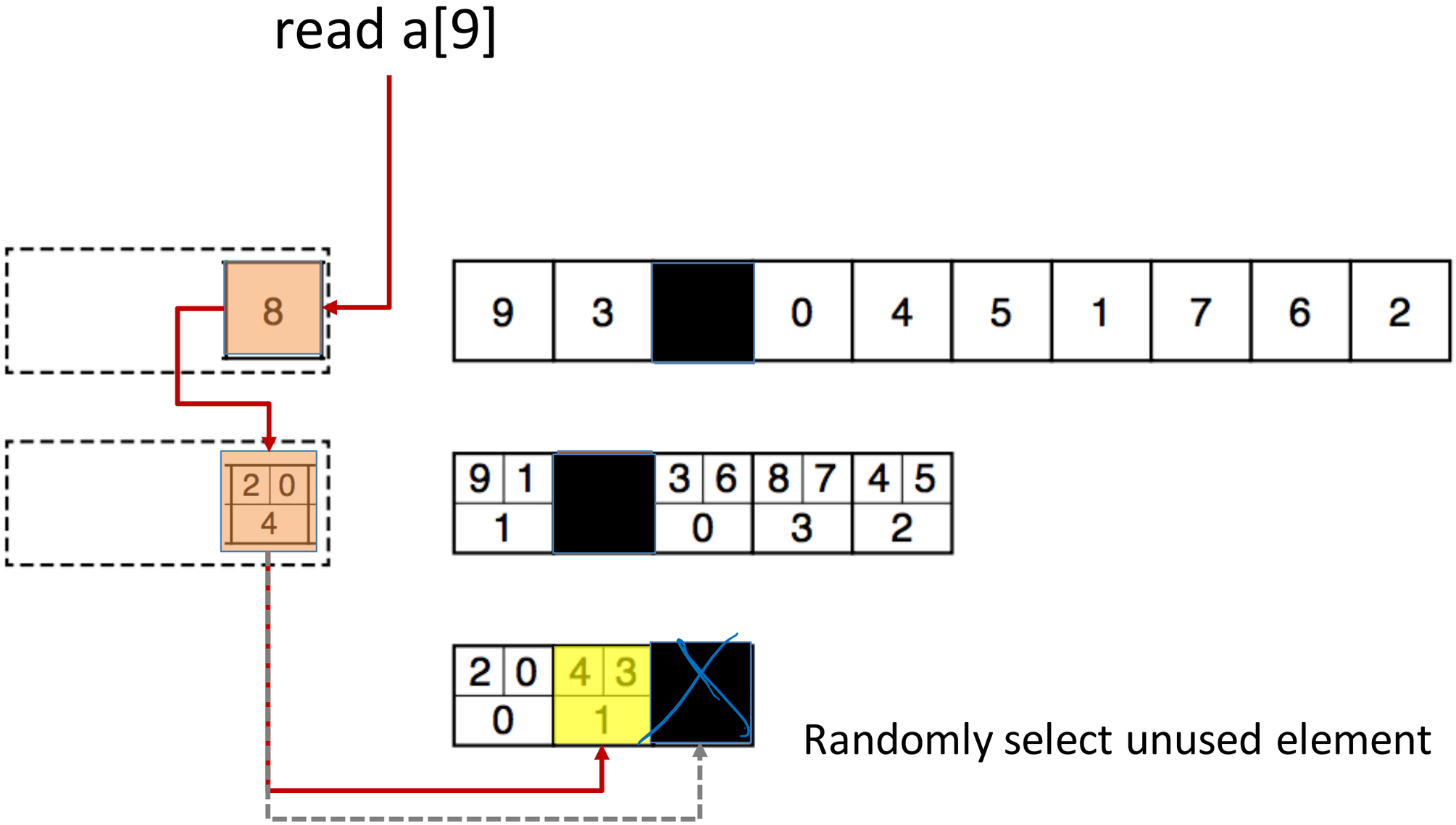


Second Access

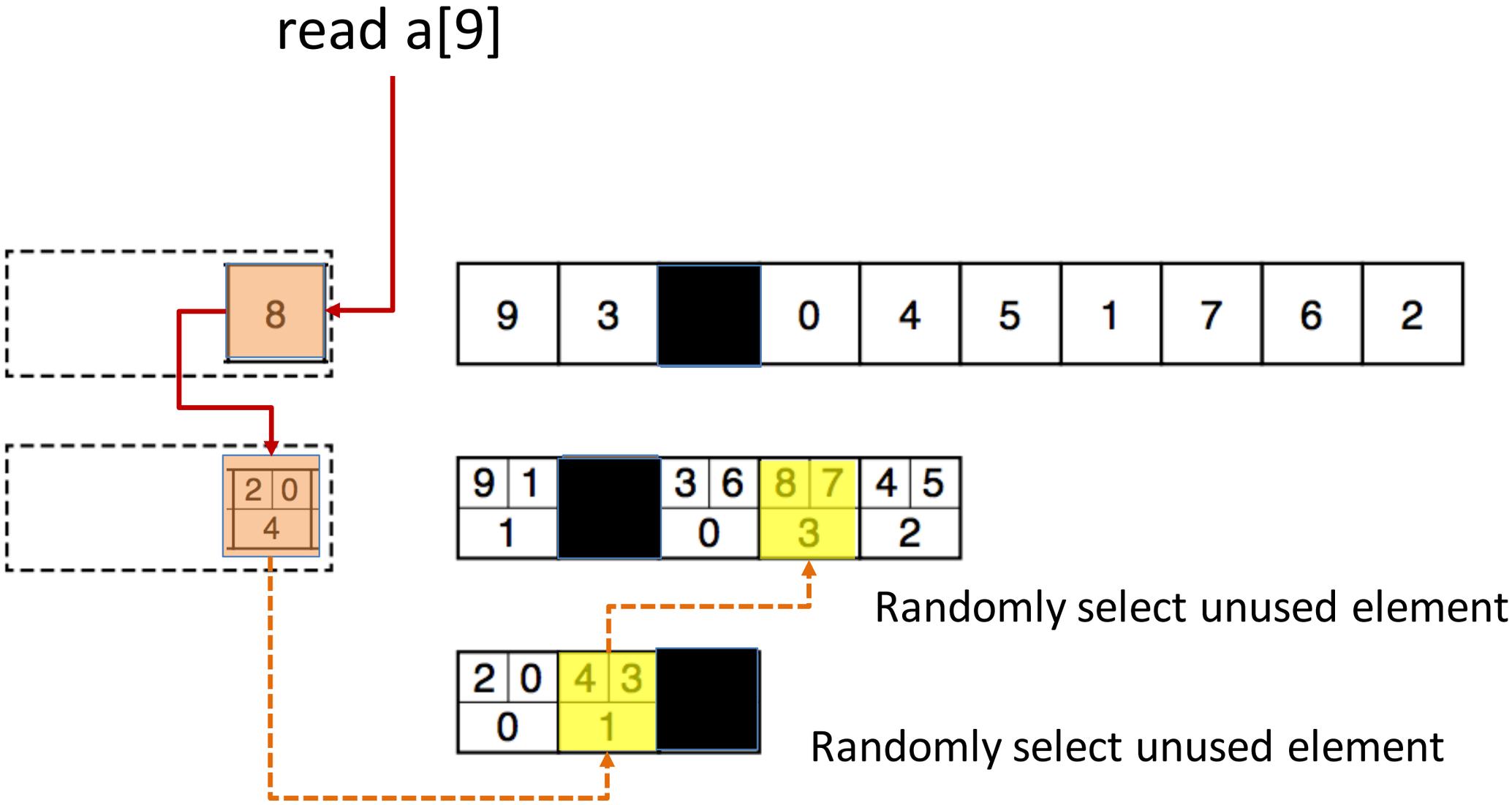
read a[9]



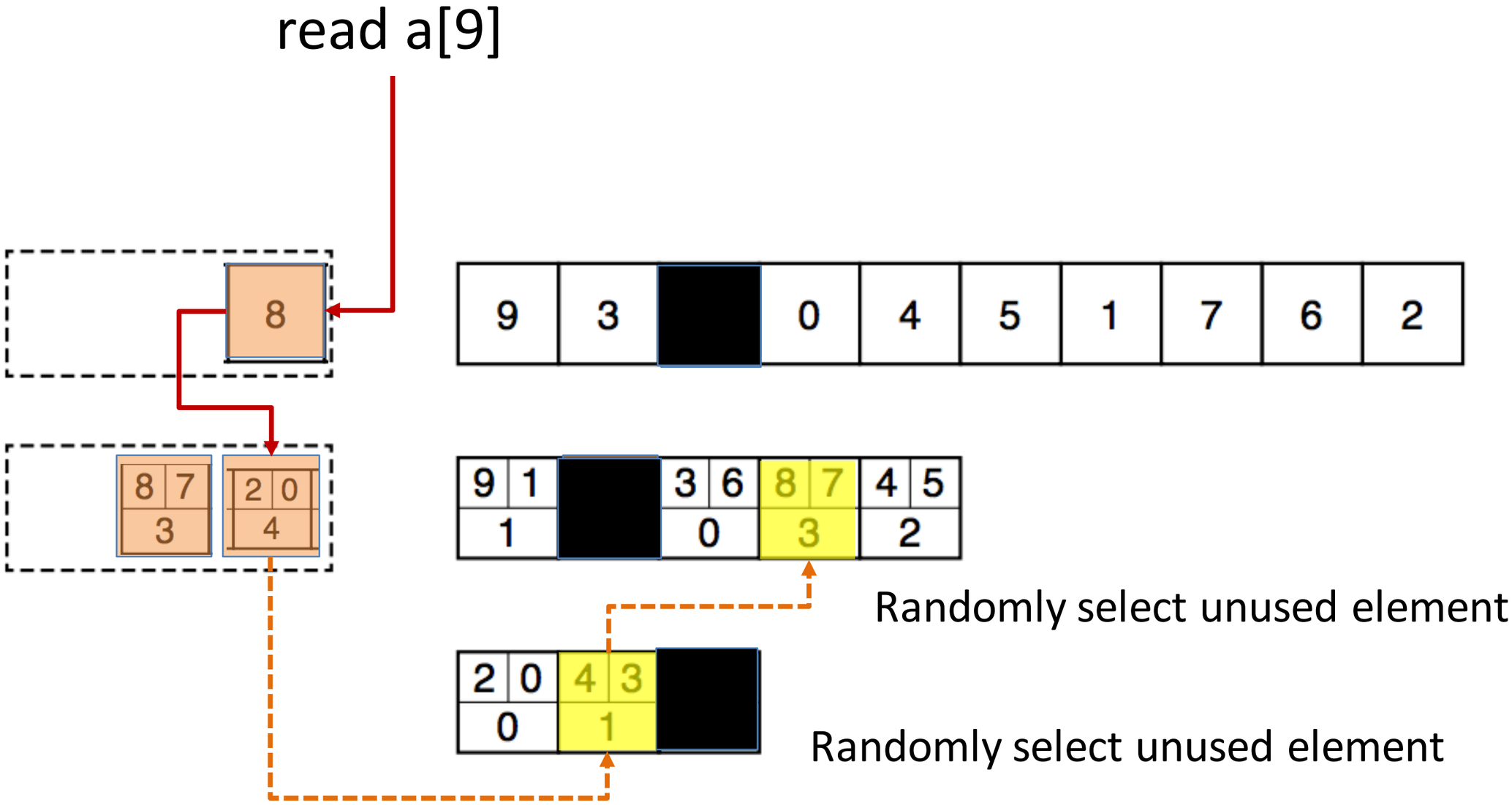
Second Access



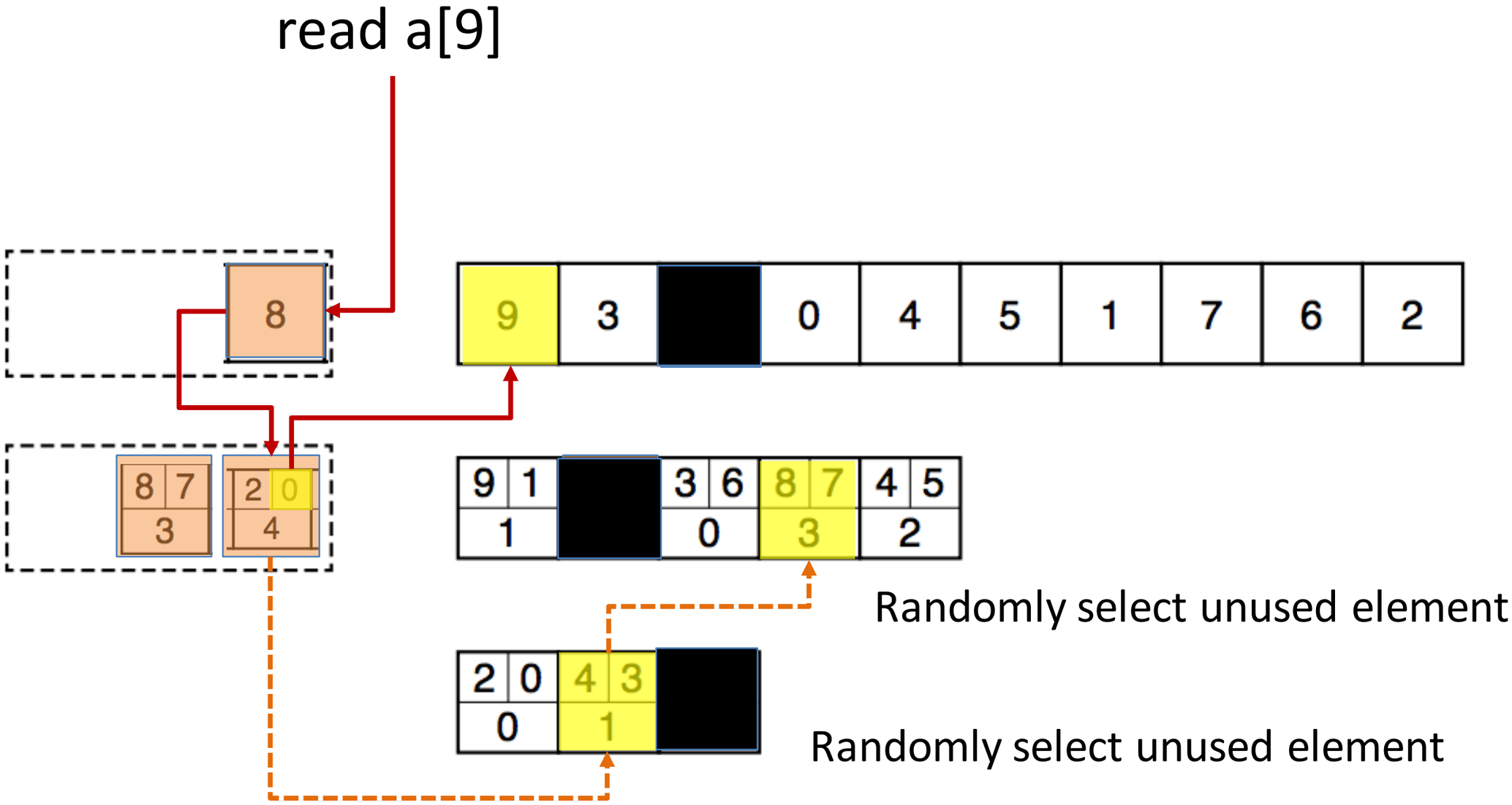
Second Access



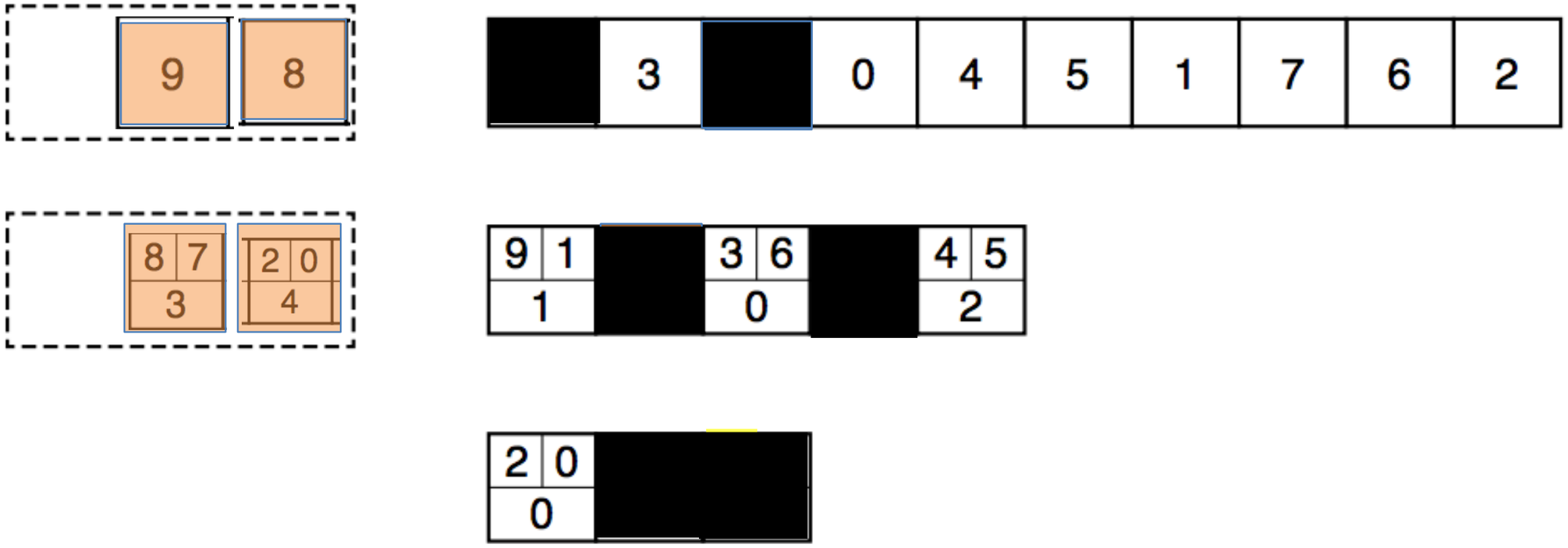
Second Access



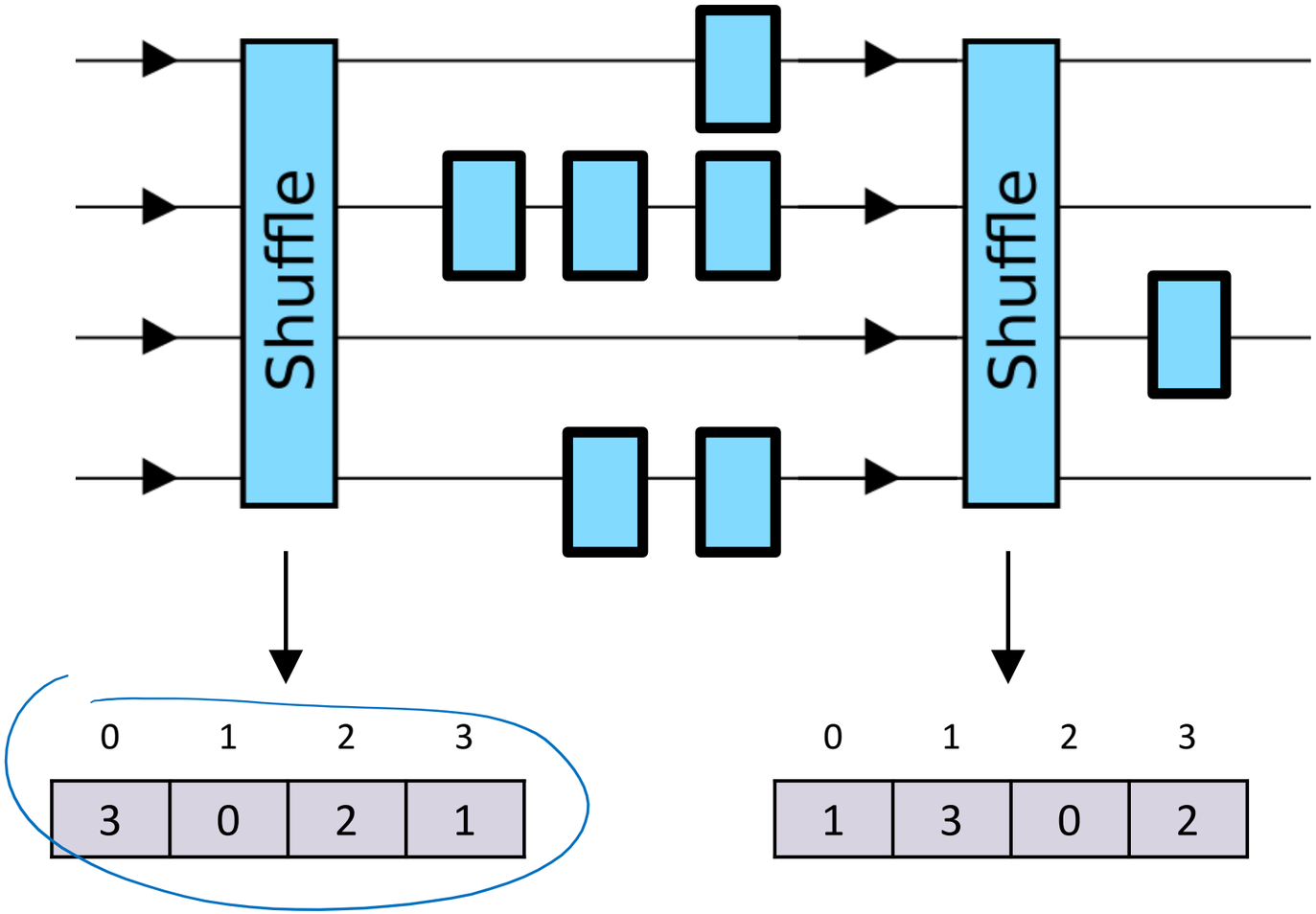
Second Access



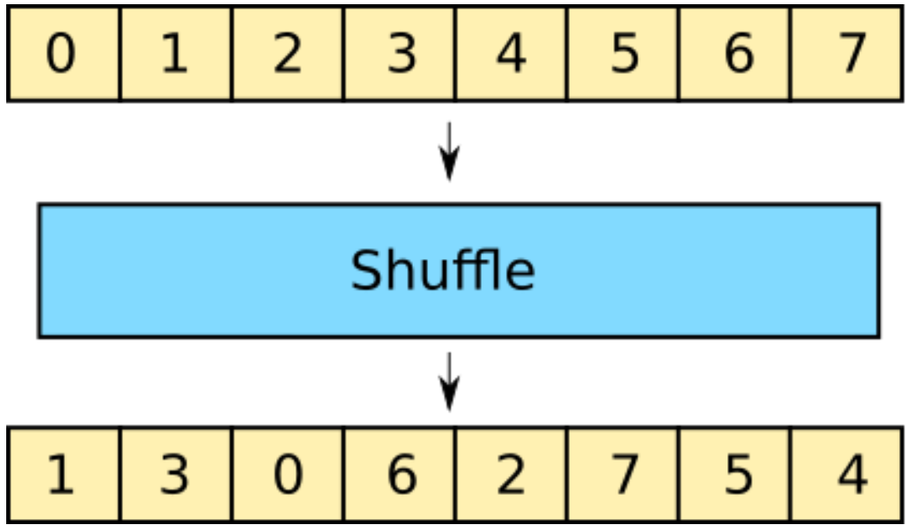
After Second Access



Position map

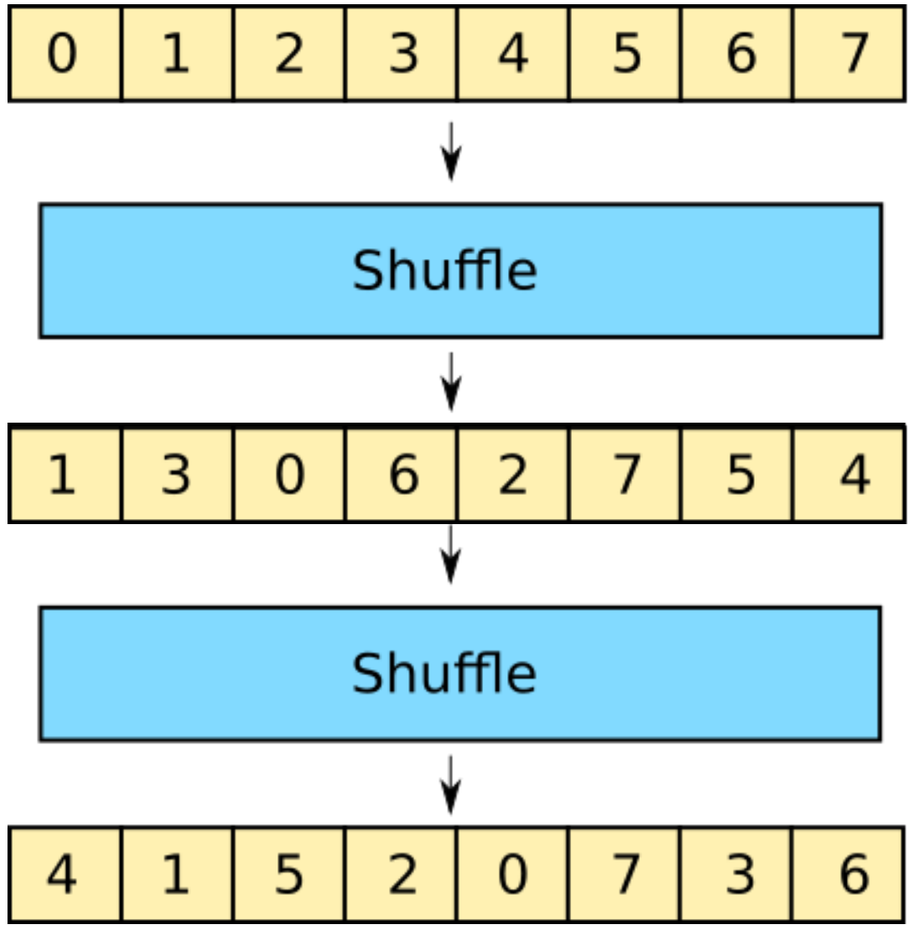


Creating position map



x	$\pi(x)$
0	2
1	0
2	4
3	1
4	7
5	6
6	3
7	5

Creating position map



x	$\pi(x)$
0	4
1	1
2	3
3	6
4	0
5	2
6	7
7	5

Inverse permutation

π_A

4	1	5	2	0	7	3	6

p

Permute

$\pi_B = \pi_A \cdot p$

Alice picks a random masking permutation

Composed permutation revealed to Bob

Inverse permutation

π_A

π_A

Permute

π_B^{-1}

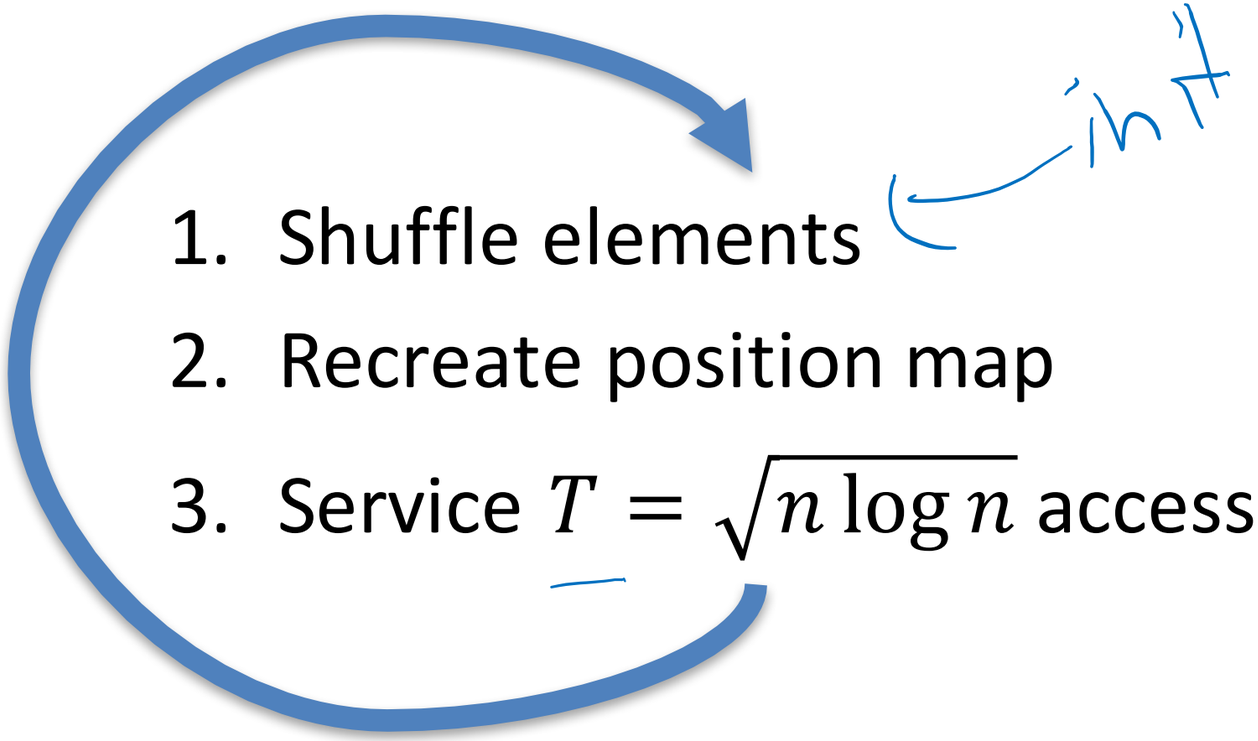
$\pi_B = \pi_A \cdot p$

Bob computes

$\pi_B^{-1} = p^{-1} \cdot \pi_A^{-1}$

$$\pi_B^{-1} \cdot \pi_A$$
$$= p^{-1} \cdot \pi_A^{-1} \cdot \pi_A$$
$$= p^{-1}$$

Scheme

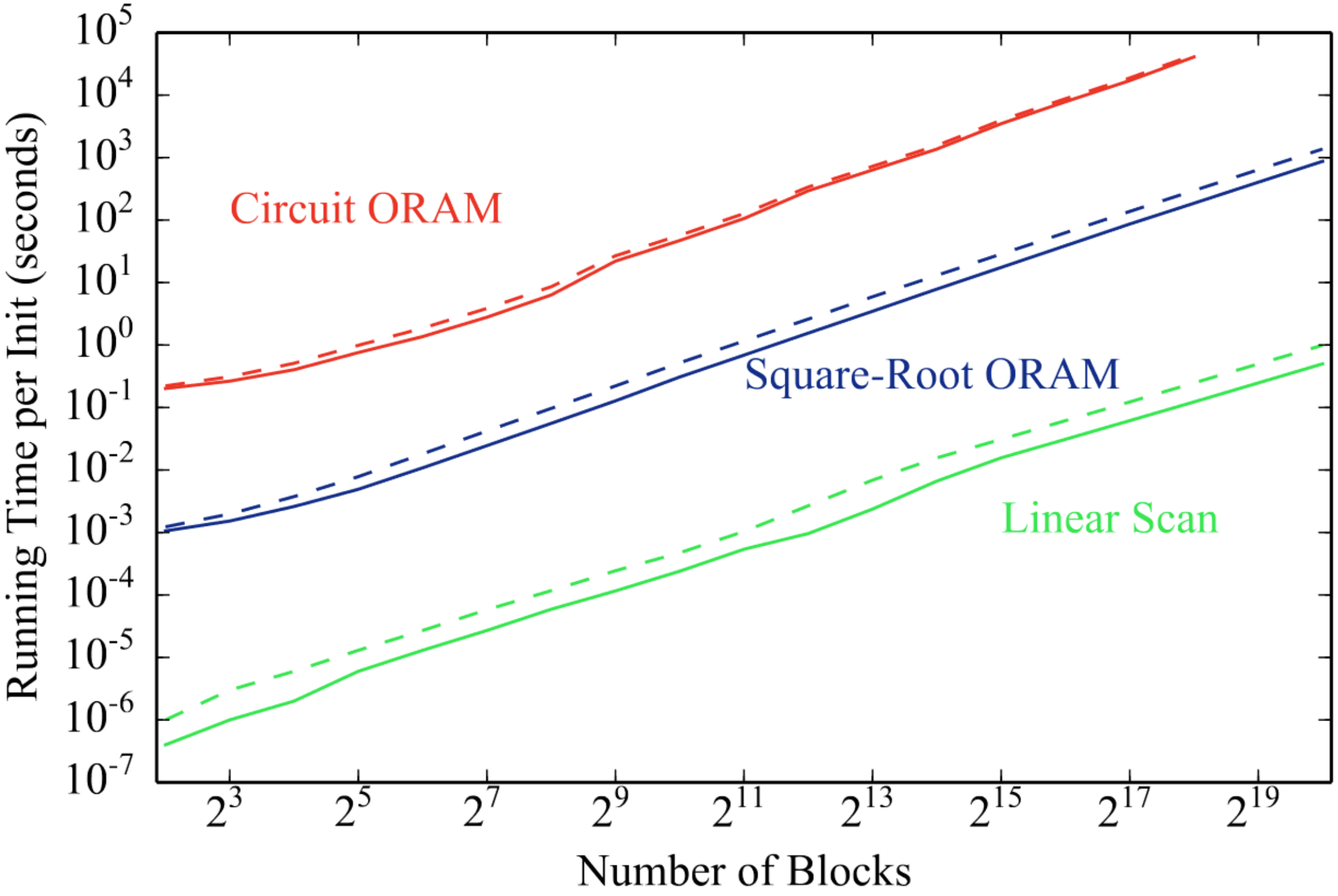


1. Shuffle elements
2. Recreate position map
3. Service $T = \sqrt{n \log n}$ accesses

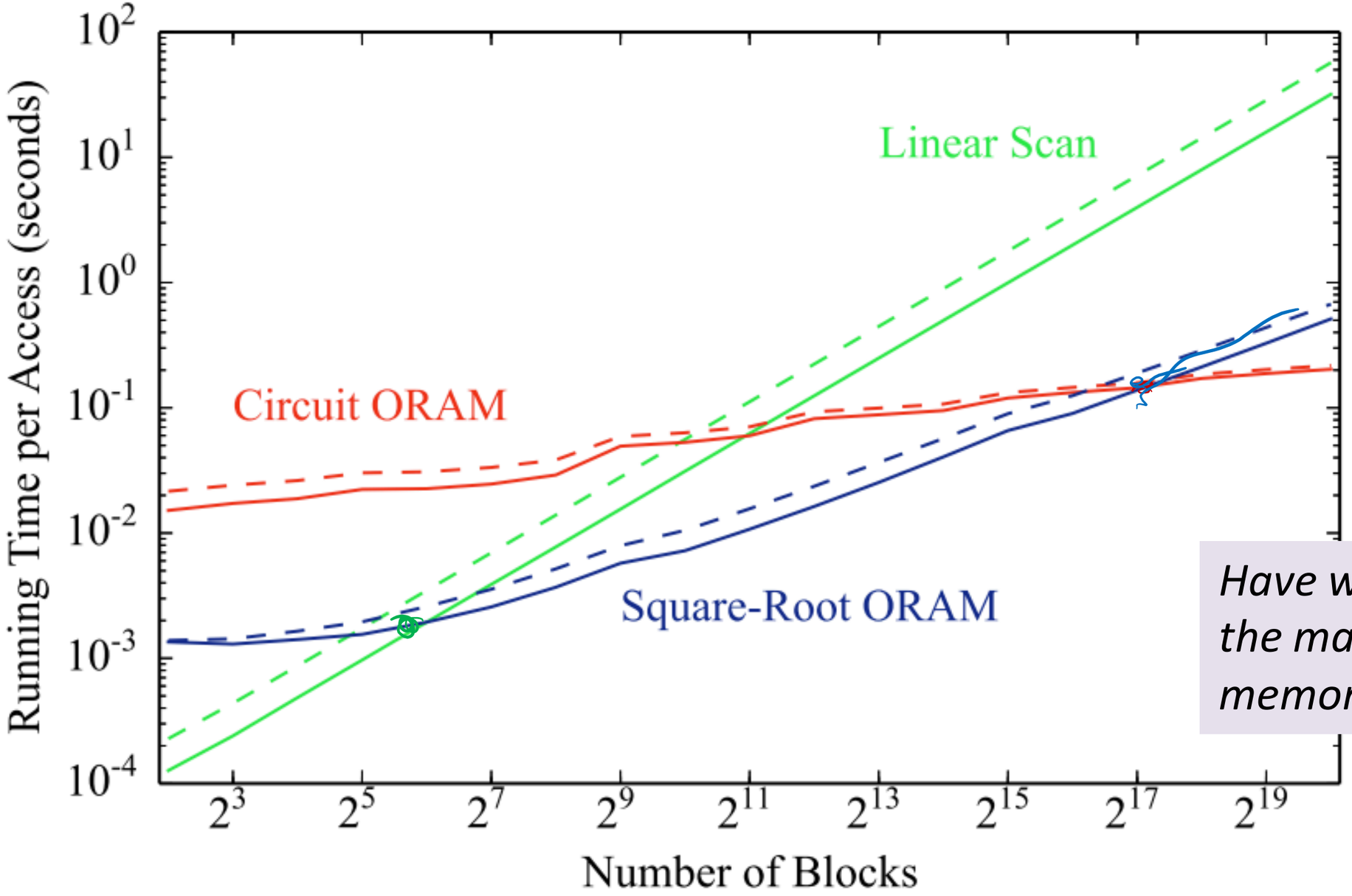
Amortized cost: $\Theta\left(B\sqrt{n \log^3 n}\right)$ per access

define Access$(\text{Oram}_0, \langle i \rangle, \Phi,)$
  $\langle \text{found} \rangle \leftarrow$ **false**
  for j **from** 0 **to** Oram.t:
    $\langle$**if**$\rangle$ $\text{Oram}_0.\text{Stash}_j.\langle \text{index} \rangle = \langle i \rangle$:
      $\langle \text{found} \rangle \leftarrow$ **true**
      $\Phi(\text{Oram}_0.\text{Stash}_j)$
  $p \leftarrow \text{GetPos}(\text{Oram}_0.\text{Oram}_1, \langle i \rangle, \langle \text{found} \rangle)$
  $\langle$**if**$\rangle$ **not** $\langle \text{found} \rangle$:
    $\Phi(\text{Oram}_0.\text{Shuffle}_p)$
  append $\text{Oram}_0.\text{Shuffle}_p$ **to** $\text{Oram}_0.\text{Stash}$
  $\text{Oram}_0.\text{Used} \leftarrow \text{Oram}_0.\text{Used} \cup \{p\}$
  $\text{Oram}_0.t \leftarrow \text{Oram}_0.t + 1$
  if $\text{Oram}_0.t = \text{Oram}_0.T$:
    for j **from** 0 **to** $\text{Oram}_0.T - 1$:
      $p' \leftarrow \text{Oram.Used}_j$
      $\text{Oram}_0.\text{Shuffle}_{p'} \leftarrow \text{Oram}_0.\text{Stash}_j$
    $\text{Oram}_0 \leftarrow \text{Initialize}(\text{Oram}_0.\text{Shuffle})$

Initialization cost



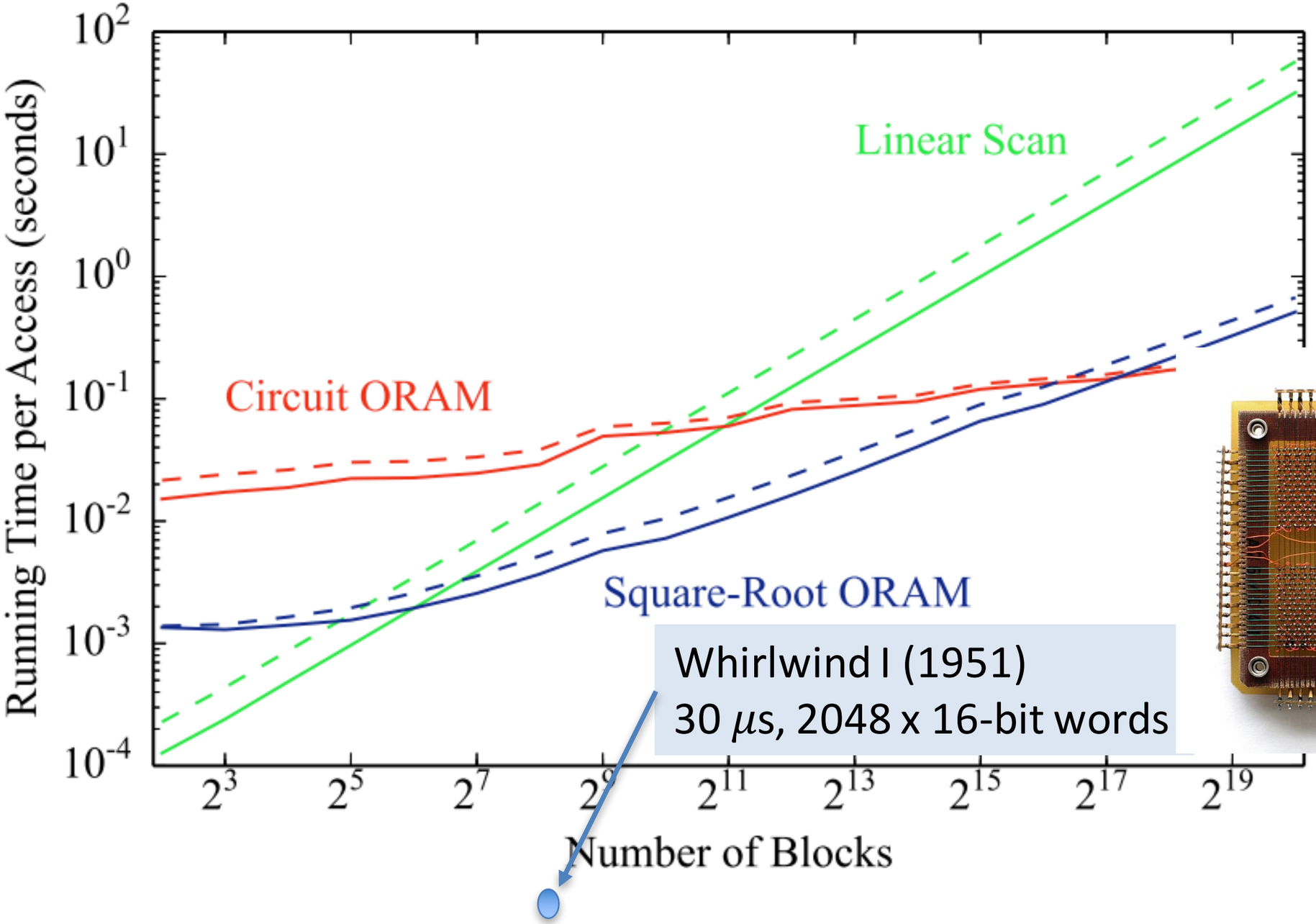
Pre-Access Cost (not counting initialization)

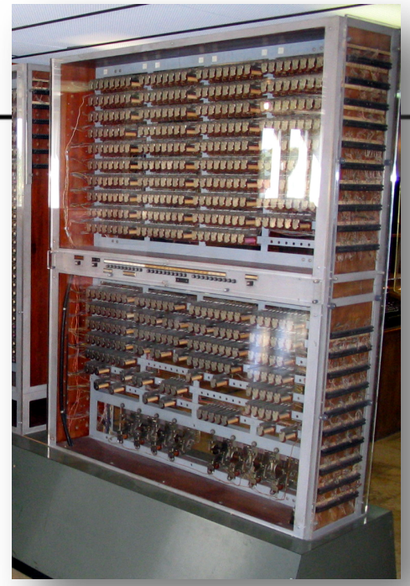


16-byte blocks

32-byte blocks

Have we reached the magnetic core memory era yet?


16-byte blocks
32-byte blocks
Linear Scan
Circuit ORAM
Square-Root ORAM
Whirlwind I (1951)
30 μs, 2048 x 16-bit words
Running Time per Access (seconds)
Number of Blocks
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^-4
2^3
2^5
2^7
2^9
2^11
2^13
2^15
2^17
2^19



Z3 (1941)

16-byte blocks

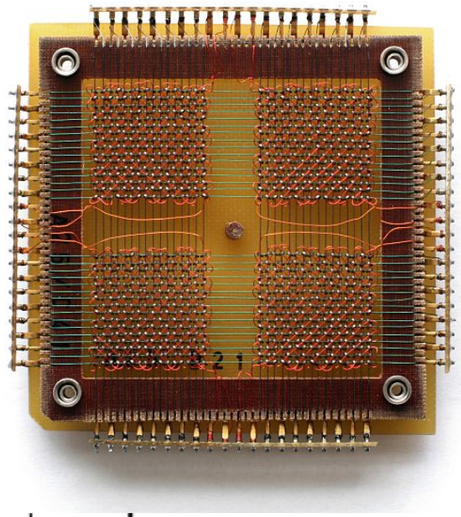
32-byte blocks

Linear Scan

Circuit ORAM

Square-Root ORAM

Whirlwind I (1951)
30 μs, 2048 x 16-bit words



Running Time per Access (seconds)

10^{2} 10^{1} 10^{0} 10^{-1} 10^{-2} 10^{-3} 10^{-4}

2^{3} 2^{5} 2^{7} 2^{9} 2^{11} 2^{13} 2^{15} 2^{17} 2^{19}

Number of Blocks

Benchmark	Parameters	Linear Scan	Square-Root ORAM	Circuit ORAM
Binary Search	1 search	1.00	10.41	3228.69
	2^5 searches	31.87	26.25	3282.40
	2^{10} searches	1019.77	824.81	5040.82
Breadth-First Search	$n = 2^2$	0.09	0.34	4.28
	$n = 2^5$	4.77	4.08	42.66
	$n = 2^{10}$	4569.31	679.63	3750.57
Gale-Shapley	2^3 pairs	-	0.51	6.57
	2^6 pairs	-	145.13	1328.50
	2^9 pairs	-	119405.	188972.
Scrypt	$N = 2^5$	4.11	3.43	34.47
	$N = 2^{10}$	1678.16	293.79	1453.85
	$N = 2^{14}$	*about 7 days*	1919.92	2846.51
	Litecoin	210.92	40.29	247.29

Wall-clock time in seconds for full protocol between two EC2 C4.2xlarge nodes (1.03 Gbps)

Benchmark	Parameters	Linear Scan	Square-Root ORAM	Circuit ORAM
Binary Search	1 search	1.00	10.41	3228.69
	2^5 searches	31.87	26.25	3282.40
	2^{10} searches	1019.77	824.81	5040.82
Breadth-First Search	$n = 2^2$	0.09	0.34	4.28
	$n = 2^5$	4.77	4.08	42.66
	$n = 2^{10}$	4569.31	679.63	3750.57
Gale-Shapley	2^3 pairs	-	0.51	6.57
	2^6 pairs	-	145.13	1328.50
	2^9 pairs	-	11[illegible]	
Scrypt	$N = 2^5$	4.11		
	$N = 2^{10}$	1678.16	[illegible]	1453.85
	$N = 2^{14}$	*about 7 days*	1919.92	2846.51
	Litecoin	210.92	40.29	247.29

~32 minutes
55,000x standard execution

Wall-clock time in seconds for full protocol between two EC2 C4.2xlarge nodes (1.03 Gbps)

Benchmark	Parameters	Linear Scan	Square-Root ORAM	Circuit ORAM
Binary Search	1 search	1.00	10.41	3228.69
	2^5 searches	31.87	26.25	3282.40
	2^{10} searches	1019.77	824.81	5040.82
Breadth-First Search	$n = 2^2$	0.09	0.34	4.28
	$n = 2^5$	4.77	4.08	42.66
	$n = 2^1$			57
Gale-Shapley	2^3 pai			57
	2^6 pairs	-	145.13	1328.50
	2^9 pairs	-	119405.	188972.
Scrypt	$N = 2^5$	4.11	3.43	34.47
	$N = 2^{10}$	1678.16	293.79	1453.85
	$N = 2^{14}$	*about 7 days*	1919.92	2846.51
	Litecoin	210.92	40.29	247.29

~33 hours ("wikipedia" version)
Improved to ~1 hour with custom structures

Wall-clock time in seconds for full protocol between two EC2 C4.2xlarge nodes (1.0 Gbps)

Open Problems

- **Scalability:** poly-logarithmic hierarchical ORAM design
- **Automatic optimization:** using custom data structures when memory access predictable
- **Stronger security models:** active security
  - All results are semi-honest model
- **Establishing Meaningful Trust**

64 KB memory
1 μs access
(~2000*x* improvement)

Collaborators

Samee Zahur
Jack Doerner
David Evans

Xiao Wang
Jonathan Katz

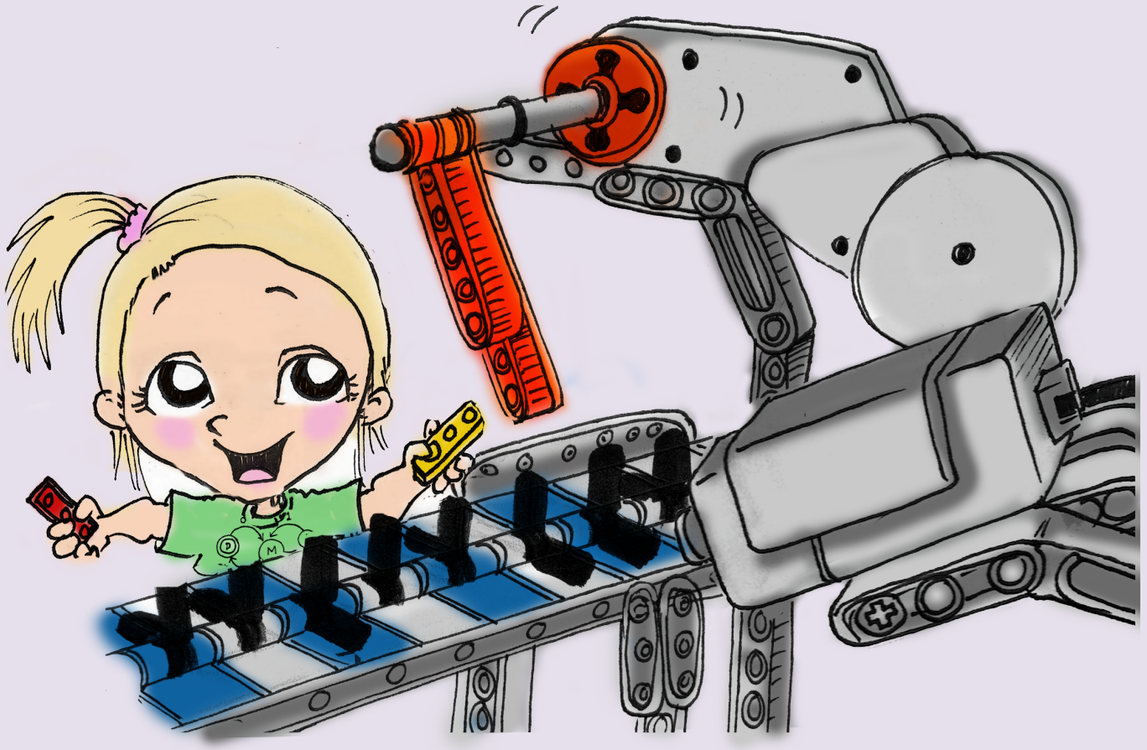
Mariana Raykova

Adrià Gascón

Code and Paper: oblivc.org/sqoram

David Evans
evans@virginia.edu

Obliv-C



www.cs.virginia.edu/evans
OblivC.org
mightBeEvil.org