Oblivious RAM

CS 598 DH

Today’s objectives

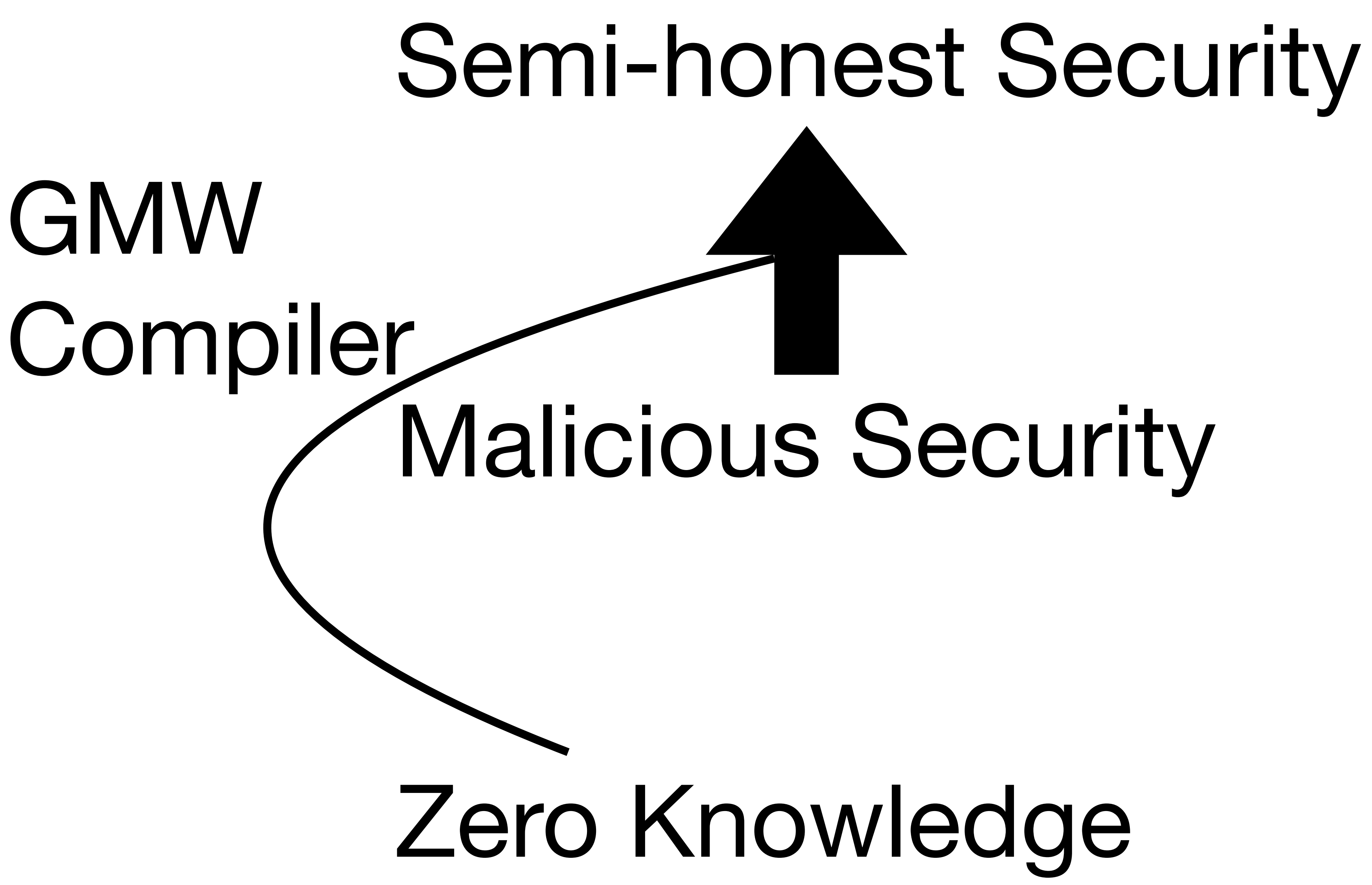
Introduce Oblivious RAM (ORAM)

Define ORAM Security

Construct non-trivial ORAM

Discuss how ORAM can be plugged into MPC

Setting



Semi-honest Security

GMW Compiler

Malicious Security

Zero Knowledge

General-Purpose Tools

GMW Protocol

- Multi-party
- Multi-round

Garbled Circuit

- Constant Round
- Two Party

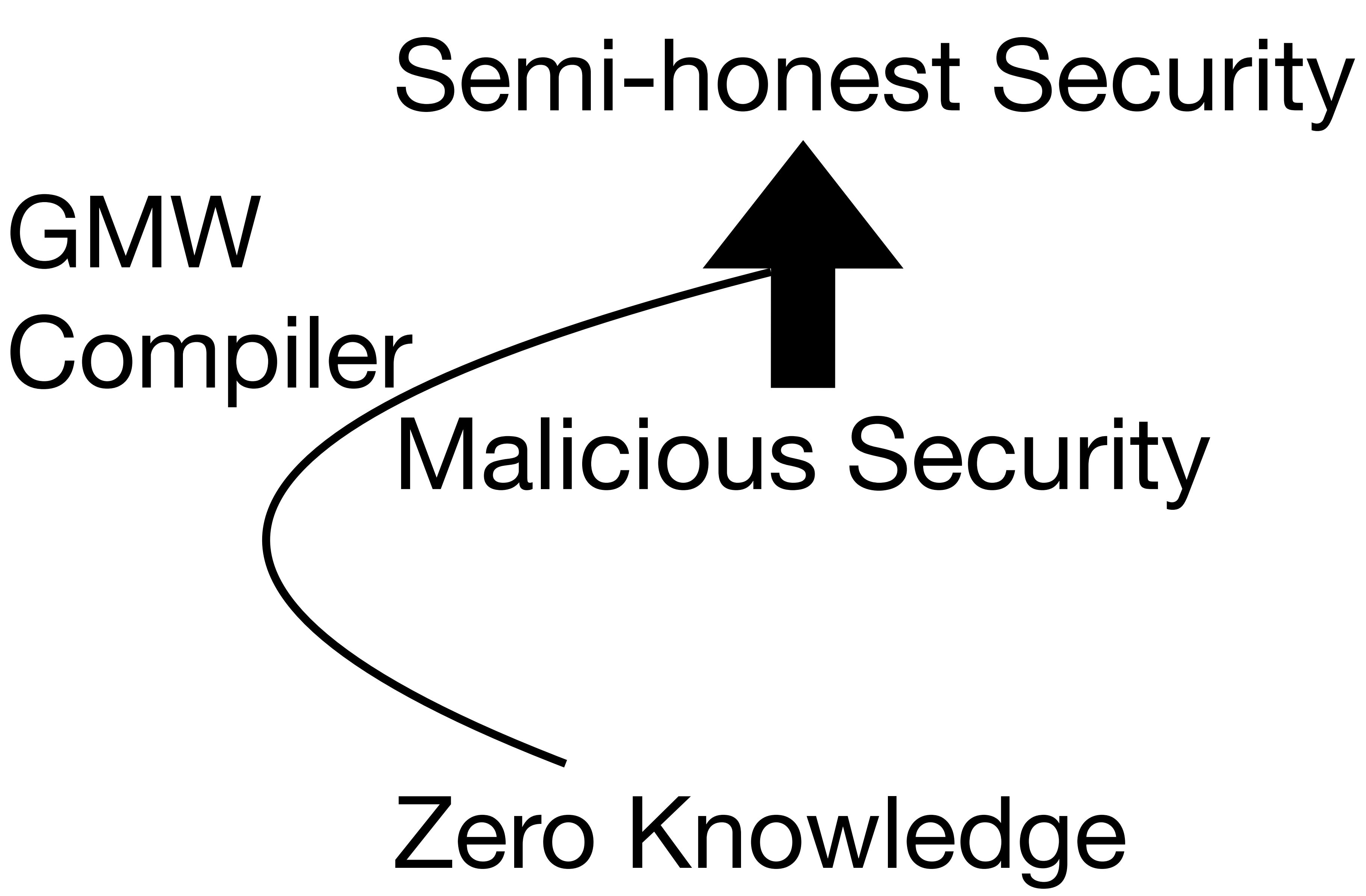
Primitives

Oblivious Transfer

Pseudorandom functions/encryption

Commitments

Setting



Semi-honest Security

GMW Compiler

Malicious Security

Zero Knowledge

General-Purpose Tools

GMW Protocol

- Multi-party
- Multi-round

Garbled Circuit

- Constant Round
- Two Party

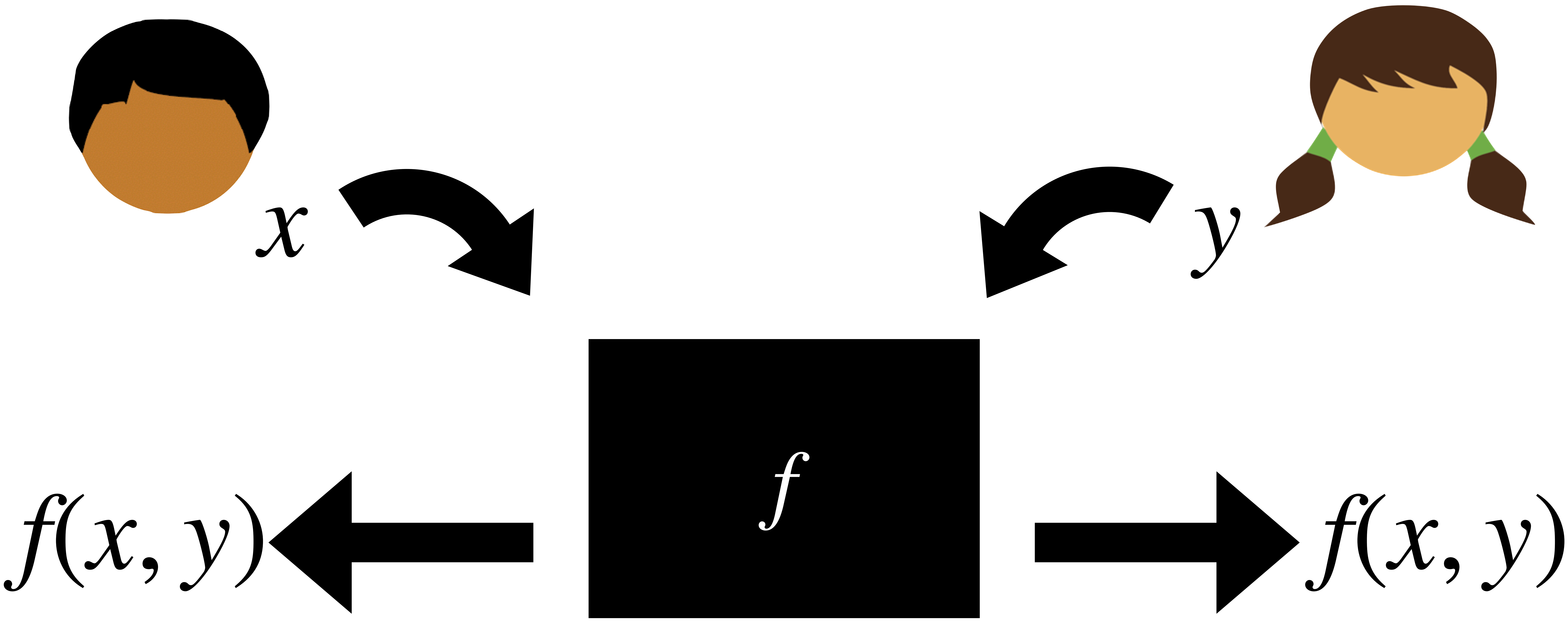
Primitives

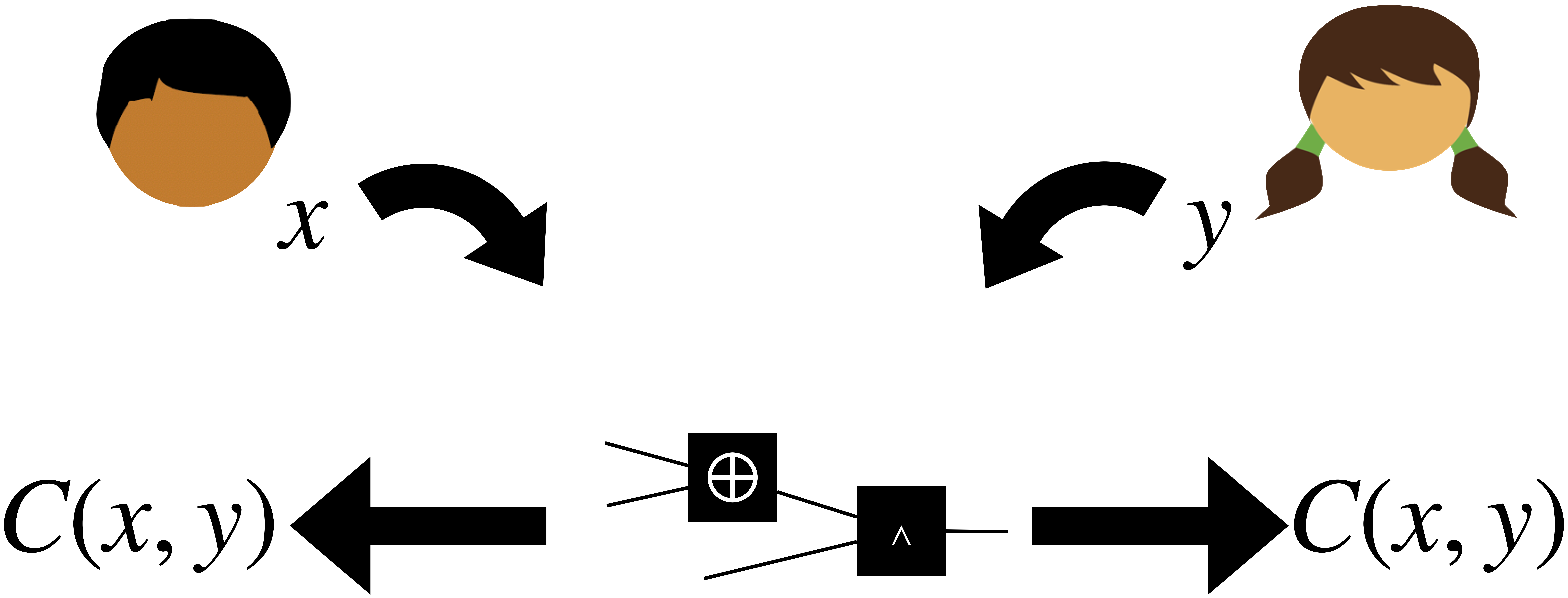
Oblivious Transfer

Pseudorandom functions/encryption

Commitments

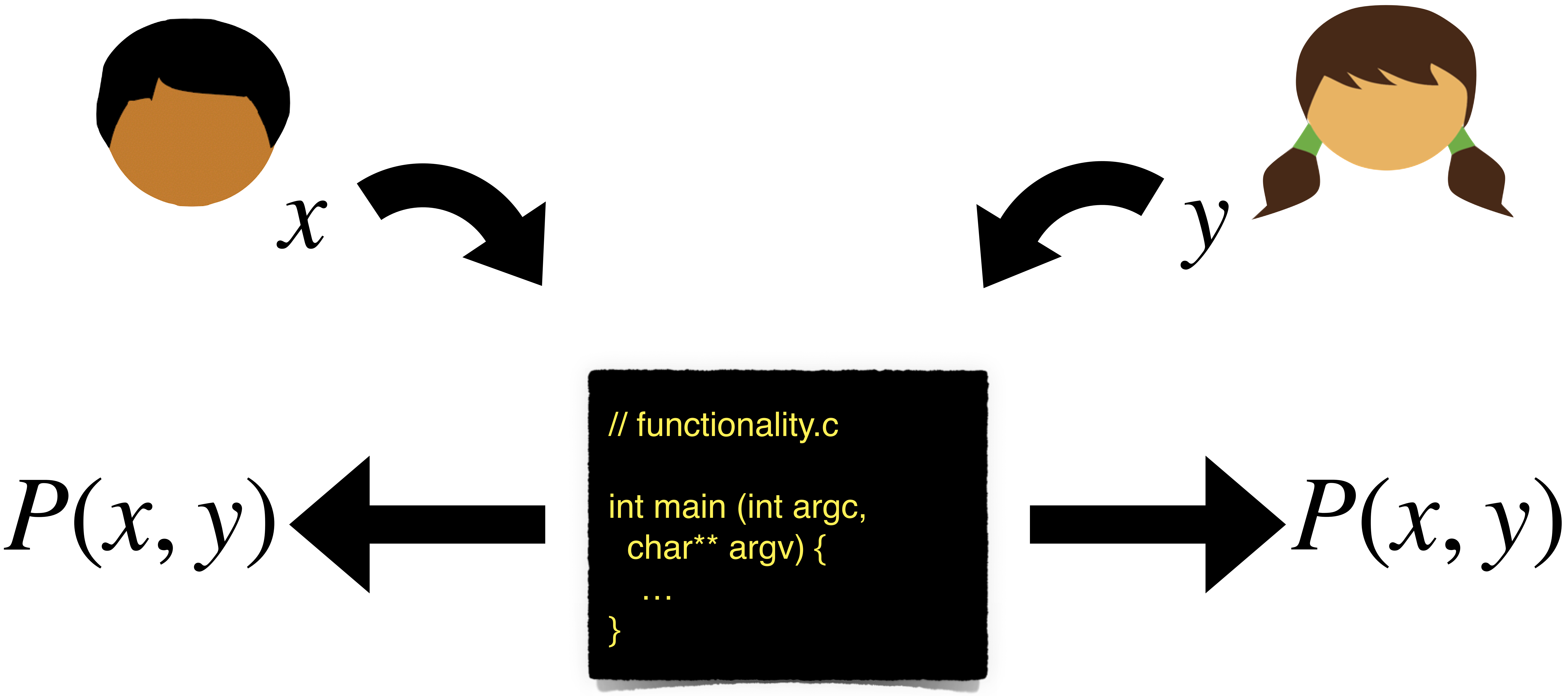
ORAM


x
y
f
$f(x, y)$
$f(x, y)$



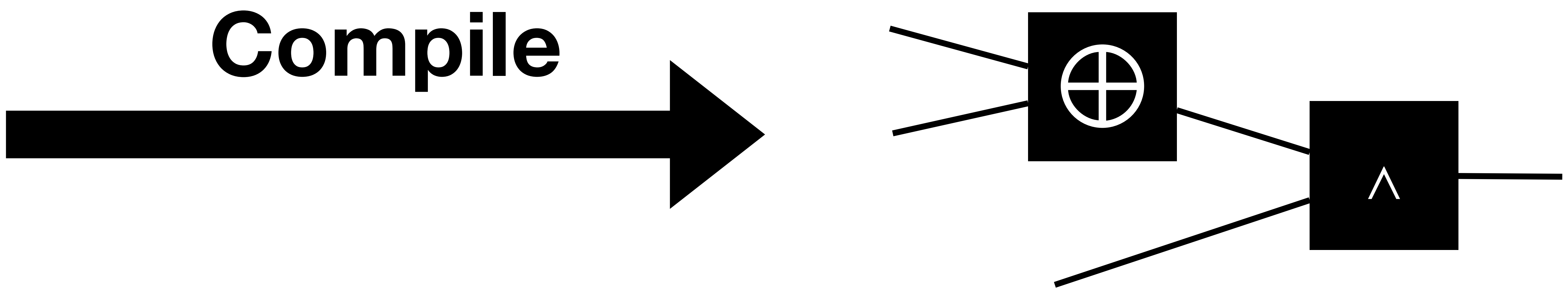
x
y
$C(x, y)$
⊕
∧
$C(x, y)$

A dream objective: Express protocol intent as a regular program

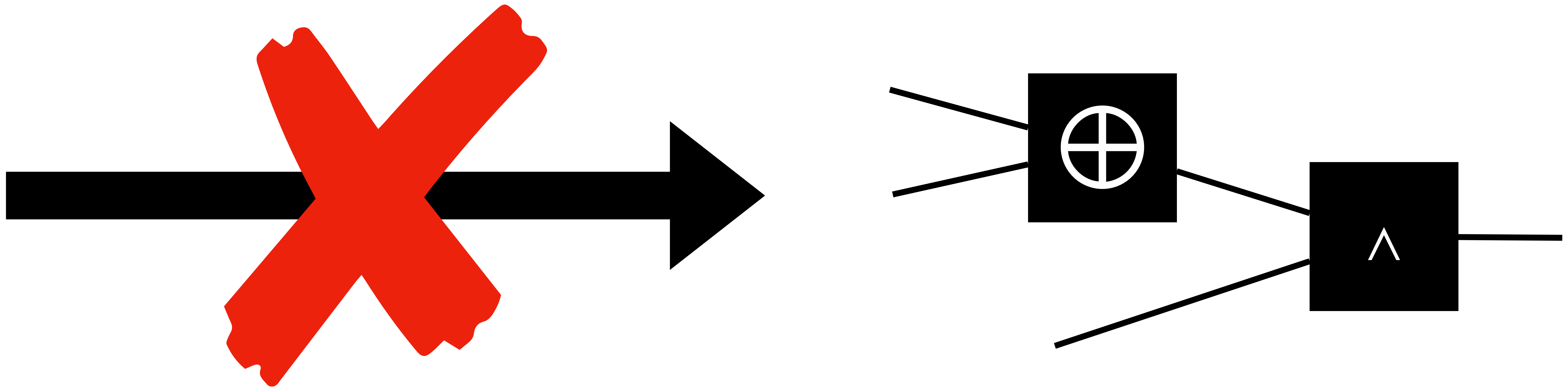


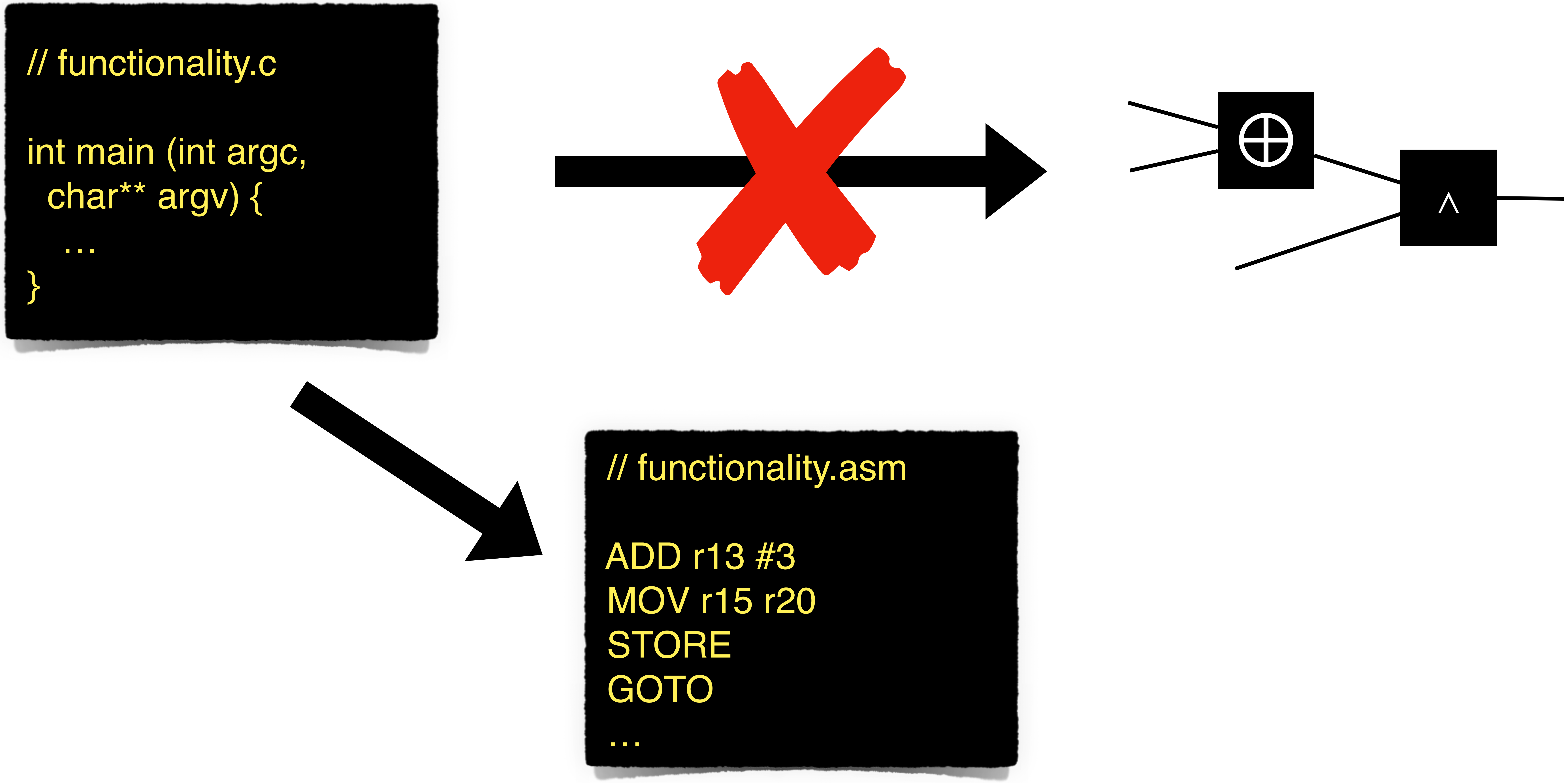
```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```

```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```


Compile

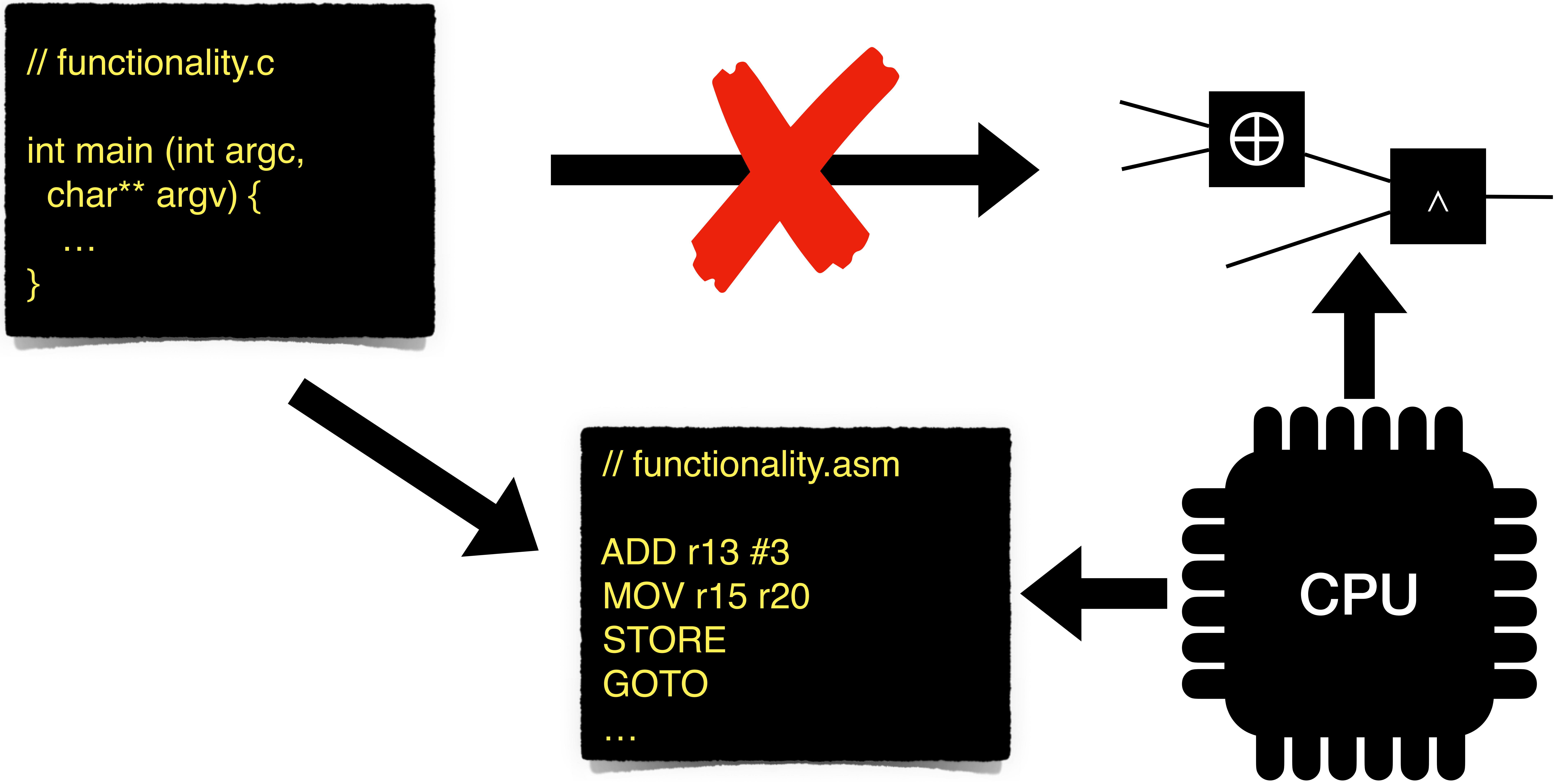
```
// functionality.c

int main (int argc,
  char** argv) {
    ...
}
```





```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```

```
// functionality.asm

ADD r13 #3
MOV r15 r20
STORE
GOTO
…
```



```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```

```
// functionality.asm

ADD r13 #3
MOV r15 r20
STORE
GOTO
…
```

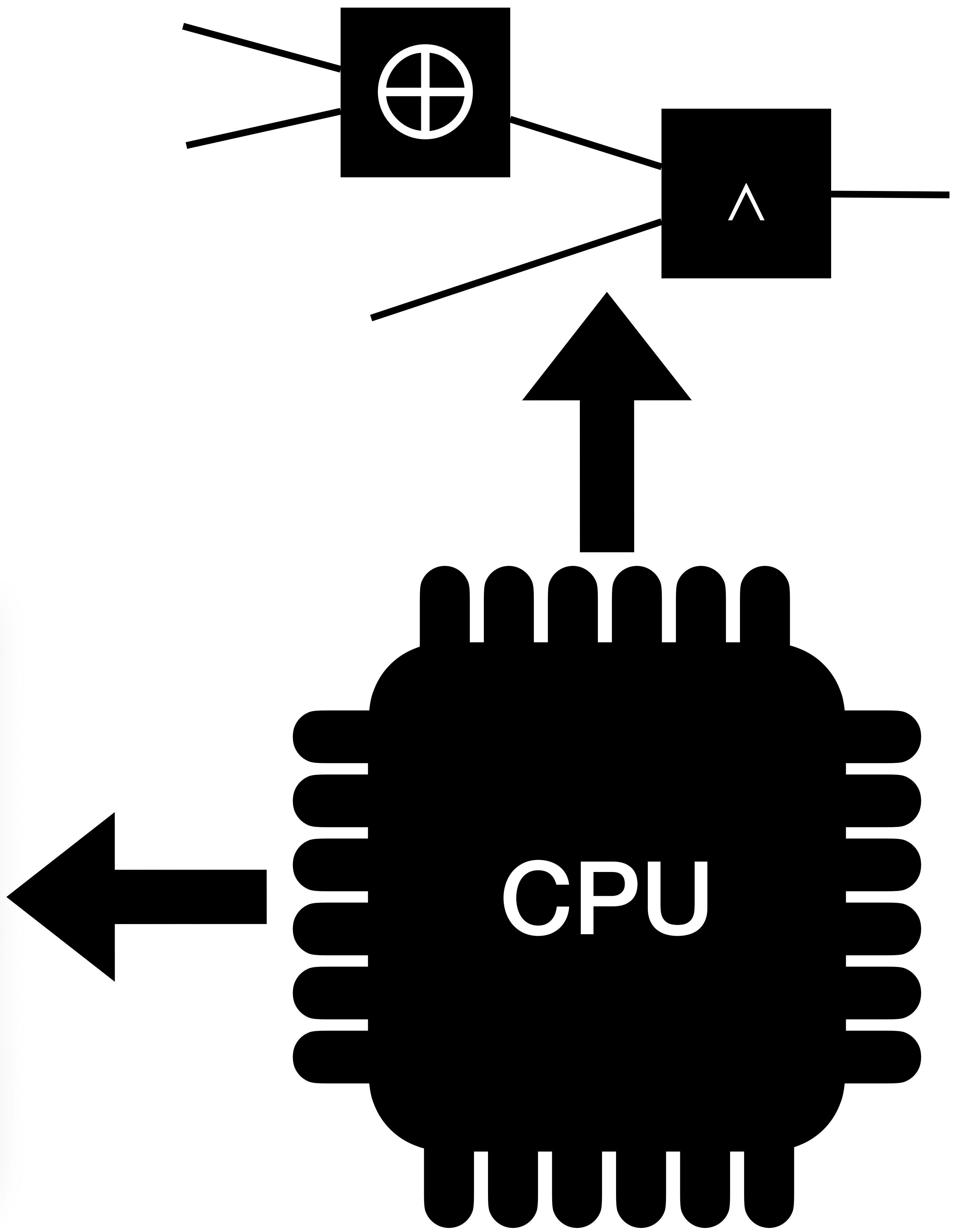
random access machine

n: size of the main memory

w: size of each memory element; **word size**

```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```

```
// functionality.asm

ADD r13 #3
MOV r15 r20
STORE
GOTO
…
```



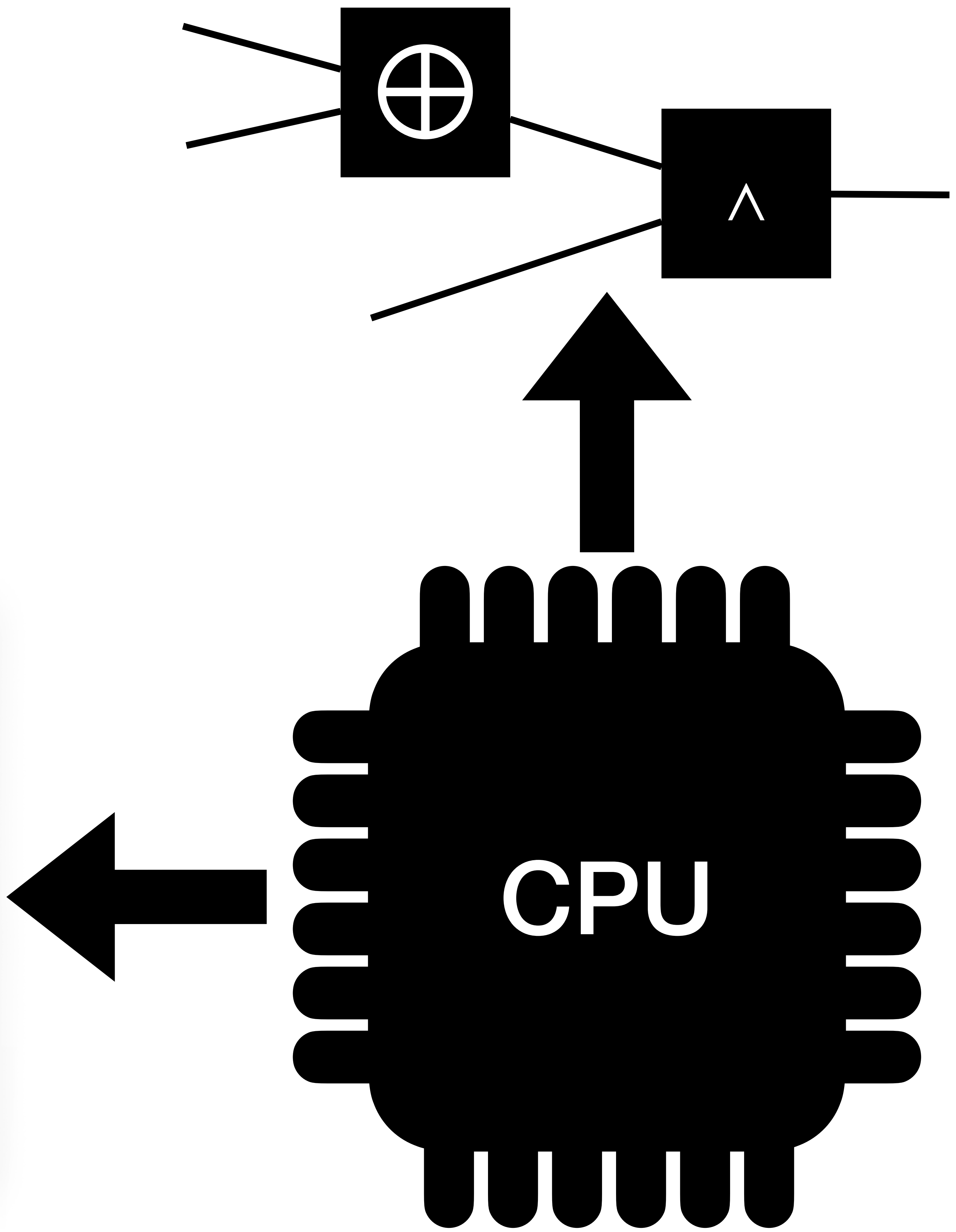
random access machine

n: size of the main memory

w: size of each memory element; **word size**

```
// functionality.c

int main (int argc,
  char** argv) {
    …
}
```

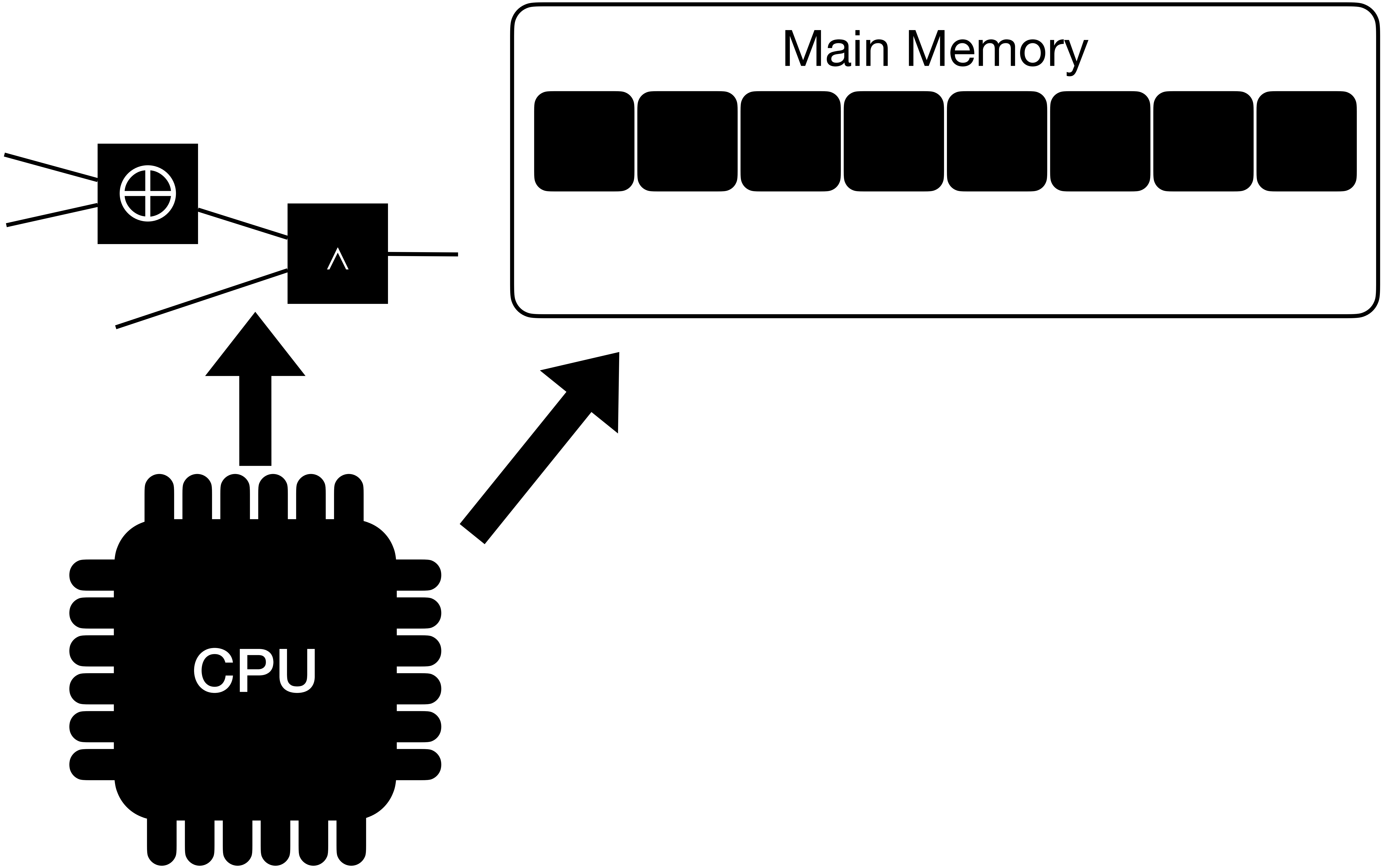
```
// functionality.asm

ADD r13 #3
MOV r15 r20
STORE
GOTO
…
```



random access machine

This works, but is prohibitively expensive because of the cost of memory access:

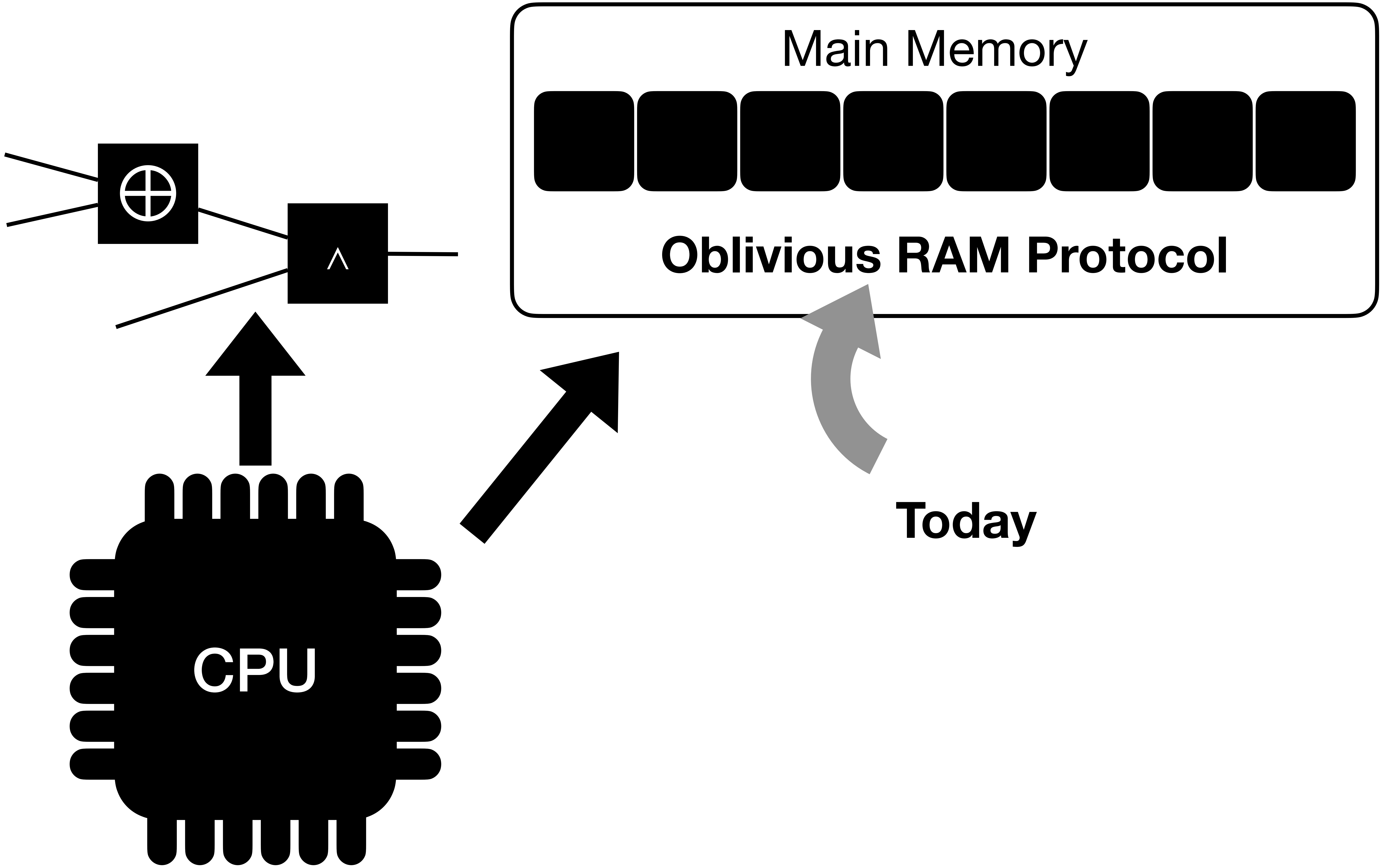
$O(w \cdot n)$ gates **per access**



Main Memory

CPU

random access machine



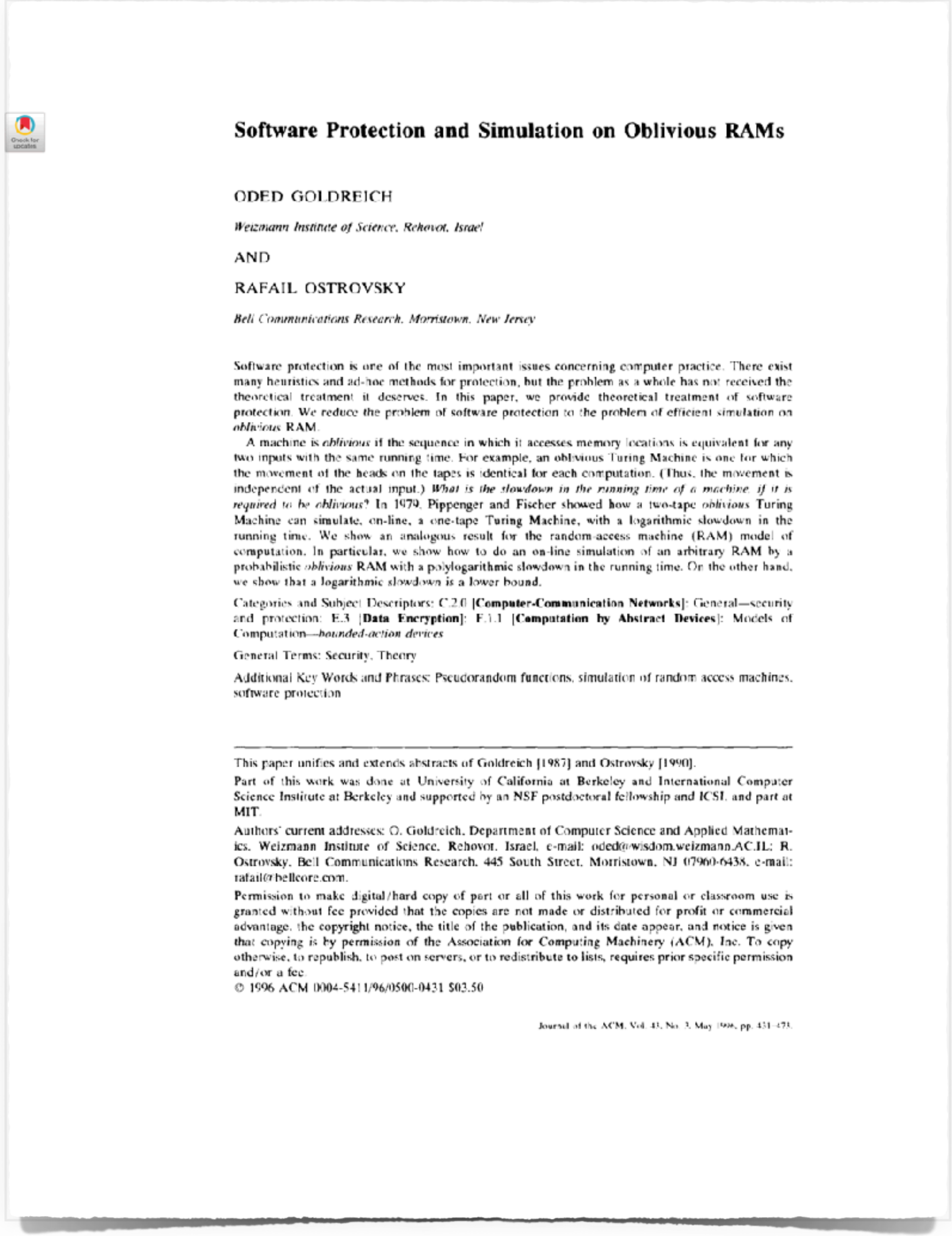
Main Memory

Oblivious RAM Protocol

Today

random access machine

Oblivious RAM (ORAM)



Software Protection and Simulation on Oblivious RAMs

ODED GOLDREICH

Weizmann Institute of Science, Rehovot, Israel

AND

RAFAIL OSTROVSKY

Bell Communications Research, Morristown, New Jersey

Software protection is one of the most important issues concerning computer practice. There exist many heuristics and ad-hoc methods for protection, but the problem as a whole has not received the theoretical treatment it deserves. In this paper, we provide theoretical treatment of software protection. We reduce the problem of software protection to the problem of efficient simulation on *oblivious* RAM.

A machine is *oblivious* if the sequence in which it accesses memory locations is equivalent for any two inputs with the same running time. For example, an oblivious Turing Machine is one for which the movement of the heads on the tapes is identical for each computation. (Thus, the movement is independent of the actual input.) *What is the slowdown in the running time of a machine, if it is required to be oblivious?* In 1979, Pippenger and Fischer showed how a two-tape *oblivious* Turing Machine can simulate, on-line, a one-tape Turing Machine, with a logarithmic slowdown in the running time. We show an analogous result for the random-access machine (RAM) model of computation. In particular, we show how to do an on-line simulation of an arbitrary RAM by a probabilistic *oblivious* RAM with a polylogarithmic slowdown in the running time. On the other hand, we show that a logarithmic slowdown is a lower bound.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—security and protection; E.3 [**Data Encryption**]; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*bounded-action devices*

General Terms: Security, Theory

Additional Key Words and Phrases: Pseudorandom functions, simulation of random access machines, software protection

This paper unifies and extends abstracts of Goldreich [1987] and Ostrovsky [1990].

Part of this work was done at University of California at Berkeley and International Computer Science Institute at Berkeley and supported by an NSF postdoctoral fellowship and ICSI, and part at MIT.

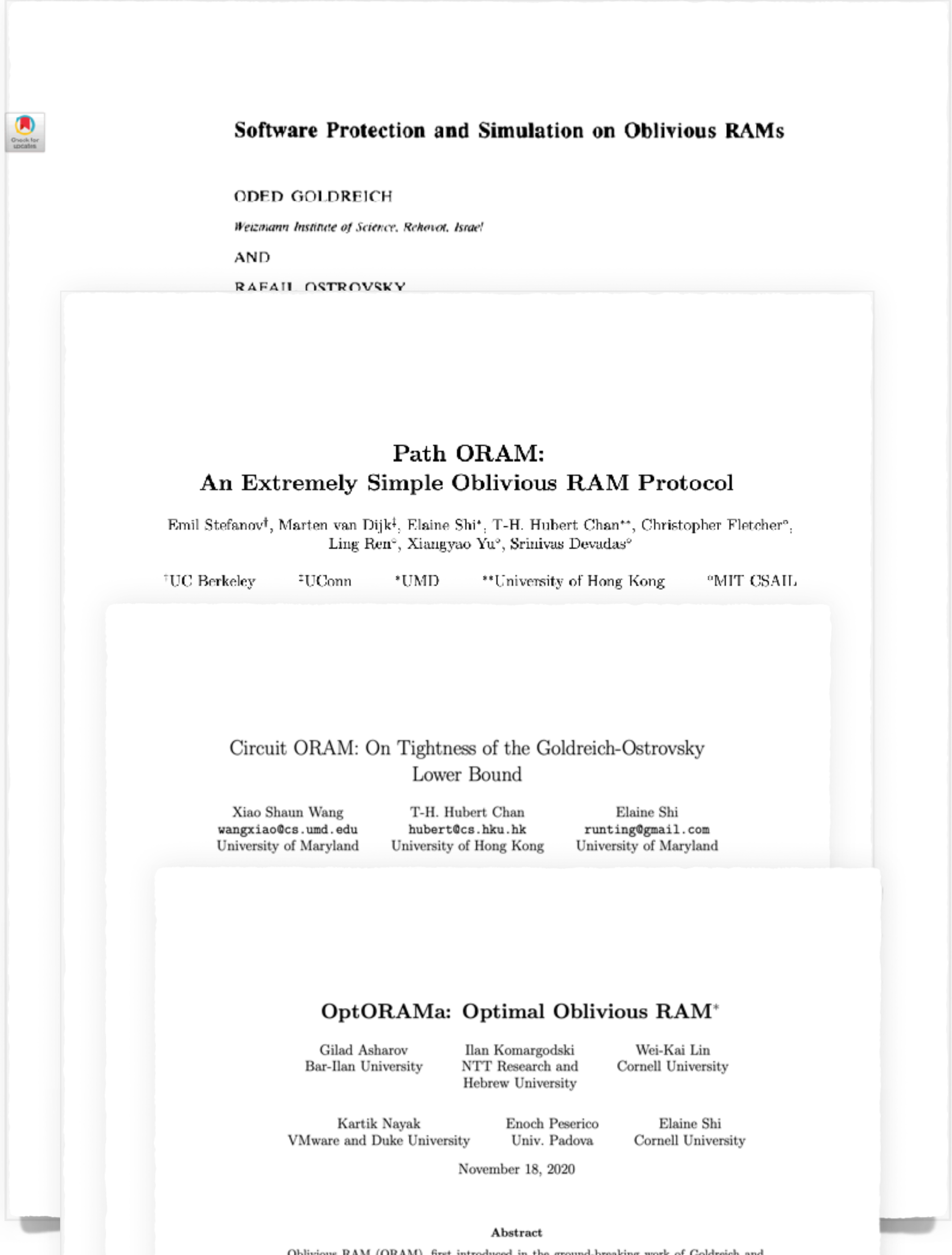
Authors' current addresses: O. Goldreich, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, e-mail: oded@wisdom.weizmann.AC.IL; R. Ostrovsky, Bell Communications Research, 445 South Street, Morristown, NJ 07960-6438, e-mail: rafail@bellcore.com.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1996 ACM 0004-5411/96/0500-0431 $03.50

Journal of the ACM, Vol. 43, No. 3, May 1996, pp. 431–473.

A protocol allowing a client to securely outsource its database to an untrusted server

Oblivious RAM (ORAM)



Software Protection and Simulation on Oblivious RAMs

ODED GOLDREICH

Weizmann Institute of Science, Rehovot, Israel

AND

RAFAIL OSTROVSKY

Path ORAM:
An Extremely Simple Oblivious RAM Protocol

Emil Stefanov[†], Marten van Dijk[‡], Elaine Shi[*], T-H. Hubert Chan[**], Christopher Fletcher[°], Ling Ren[°], Xiangyao Yu[°], Srinivas Devadas[°]

[†]UC Berkeley [‡]UConn [*]UMD [**]University of Hong Kong [°]MIT CSAIL

Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound

Xiao Shaun Wang
wangxiao@cs.umd.edu
University of Maryland

T-H. Hubert Chan
hubert@cs.hku.hk
University of Hong Kong

Elaine Shi
runting@gmail.com
University of Maryland

OptORAMa: Optimal Oblivious RAM*

Gilad Asharov
Bar-Ilan University

Ilan Komargodski
NTT Research and Hebrew University

Wei-Kai Lin
Cornell University

Kartik Nayak
VMware and Duke University

Enoch Peserico
Univ. Padova

Elaine Shi
Cornell University

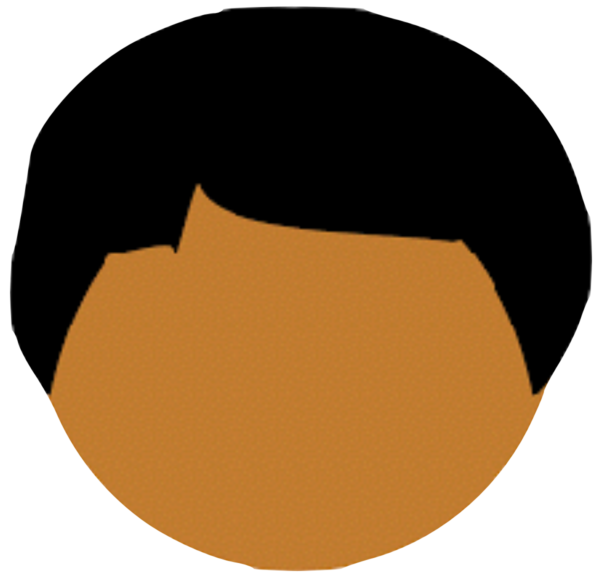
November 18, 2020

Abstract

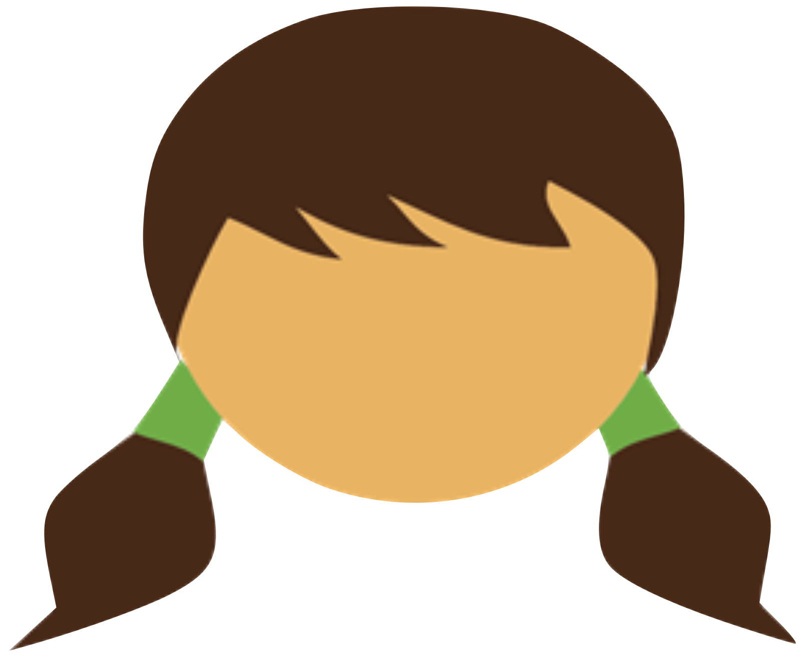
Oblivious RAM (ORAM), first introduced in the ground-breaking work of Goldreich and

A protocol allowing a client to securely outsource its database to an untrusted server

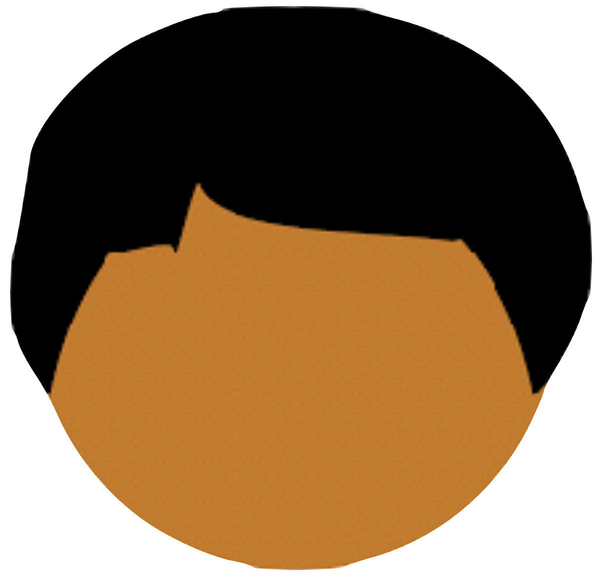
ORAM is its own research area with a large (and growing) body of work



Server S



Client C

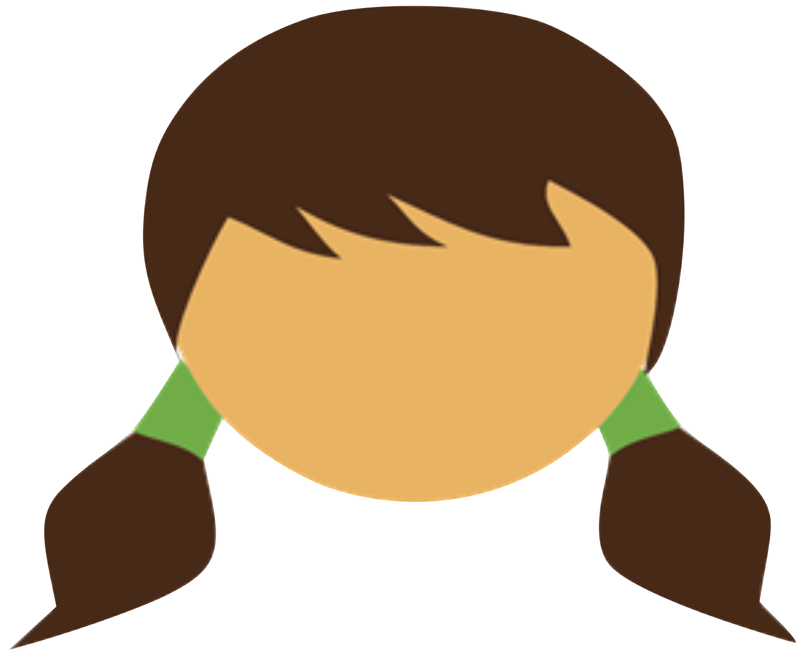


Server S

Powerful (i.e. has lots of memory)

Untrusted (semi-honest)

Has no input

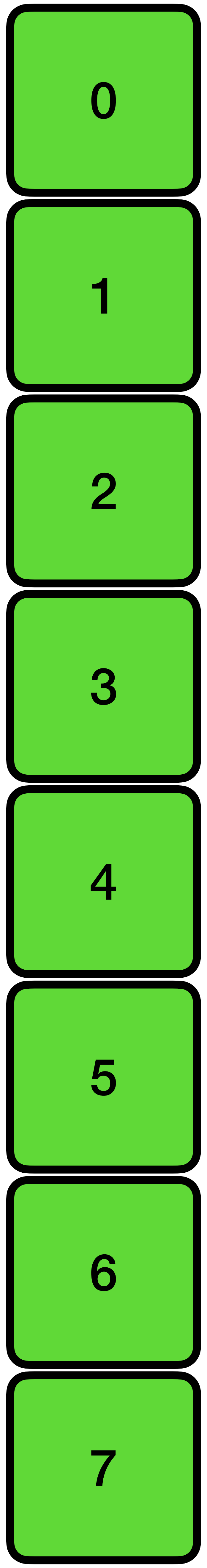


Client C

Weak (has only enough space for a few memory elements)

Wishes to repeatedly access its outsourced database

(Logical) Memory



0

1

2

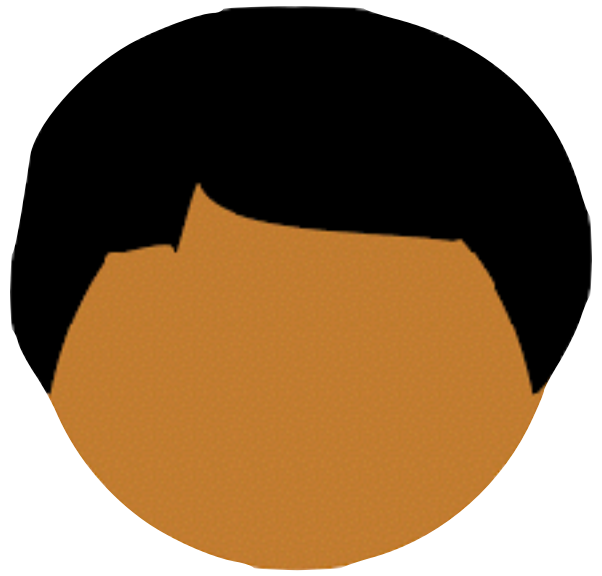
3

4

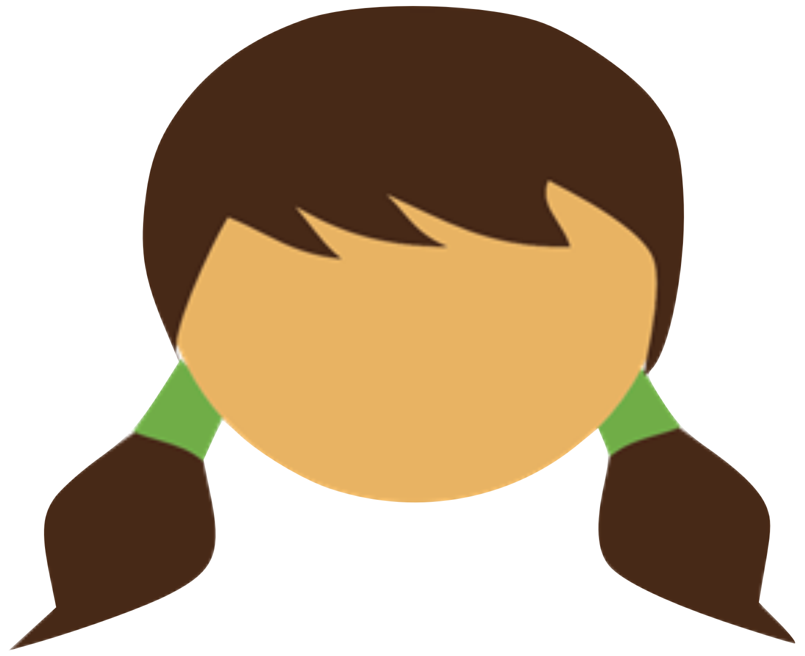
5

6

7

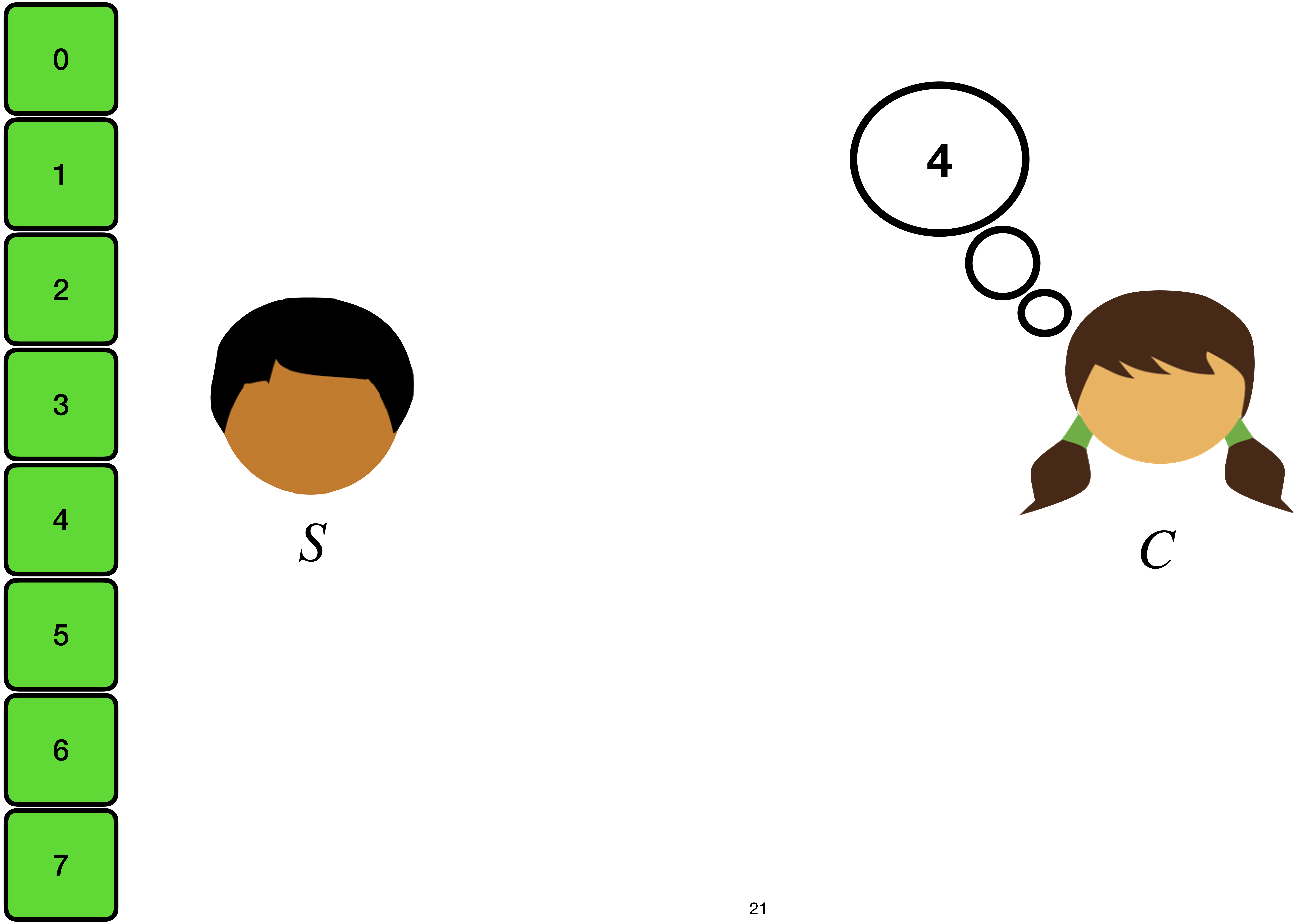


S

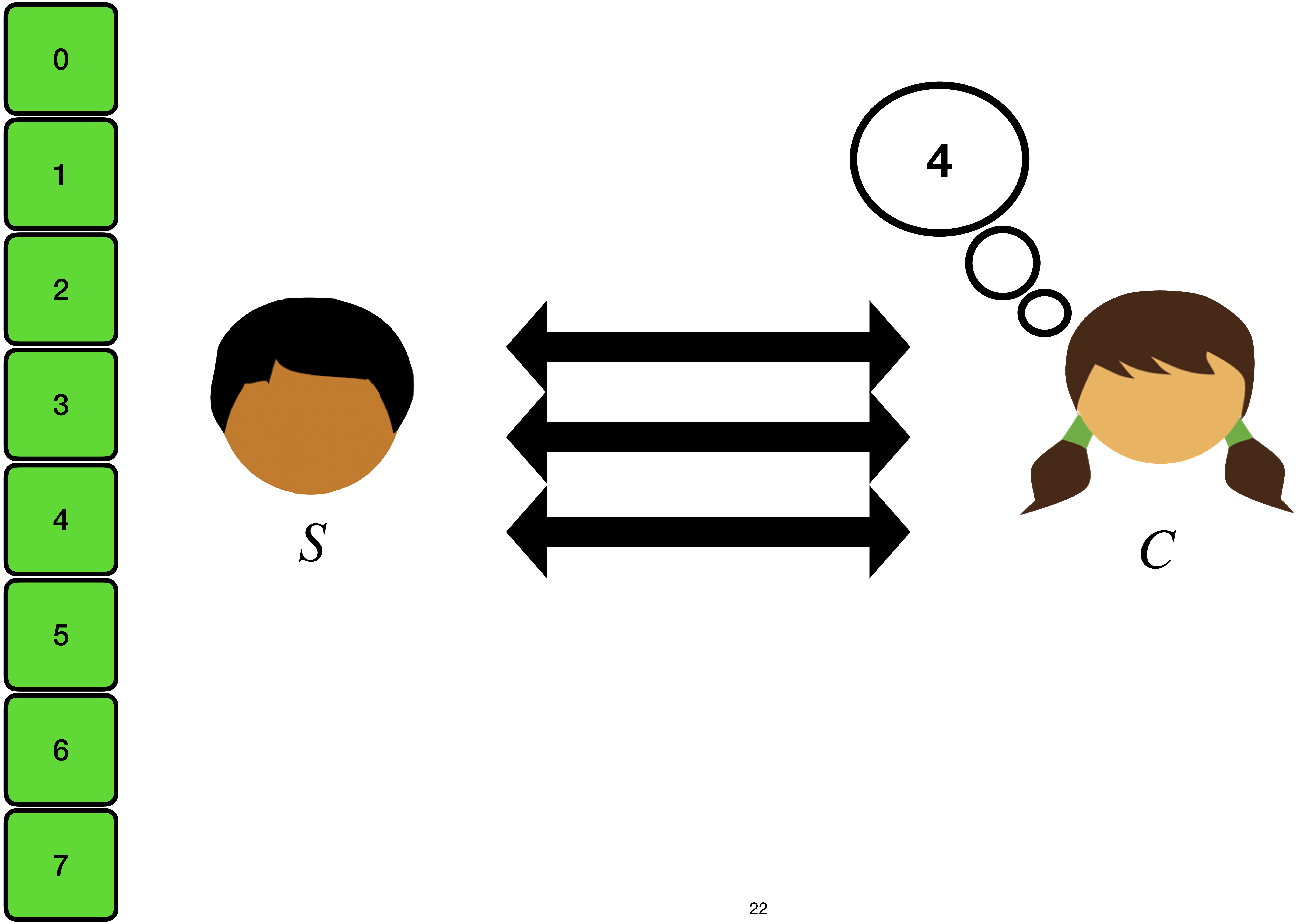


C

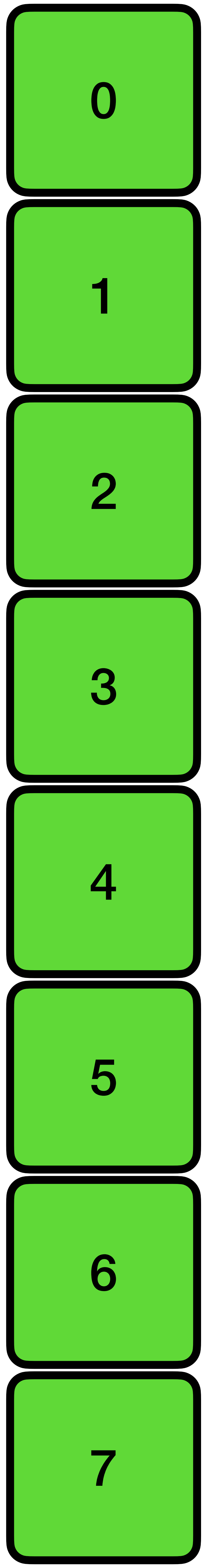
(Logical) Memory

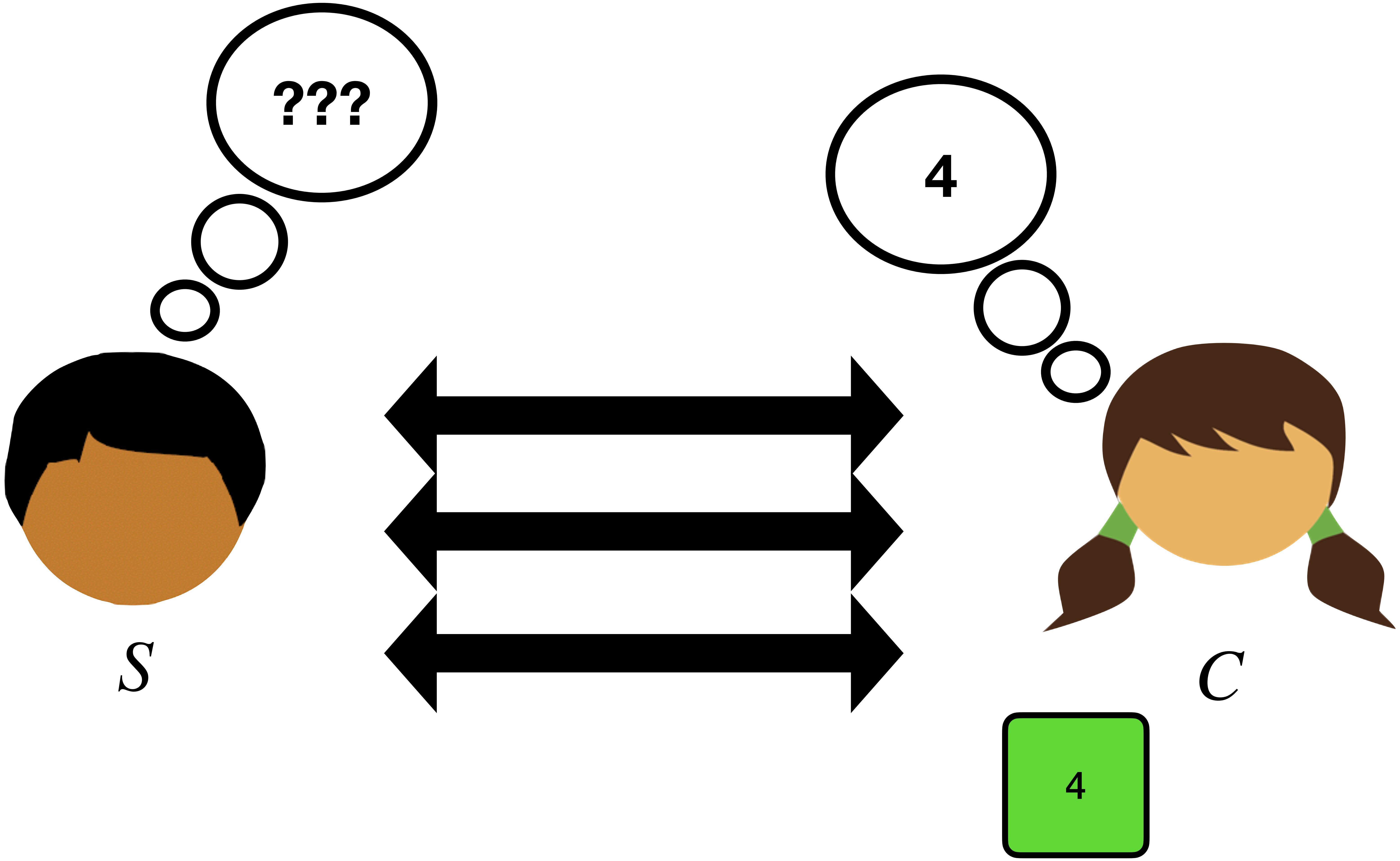


(Logical) Memory



(Logical) Memory

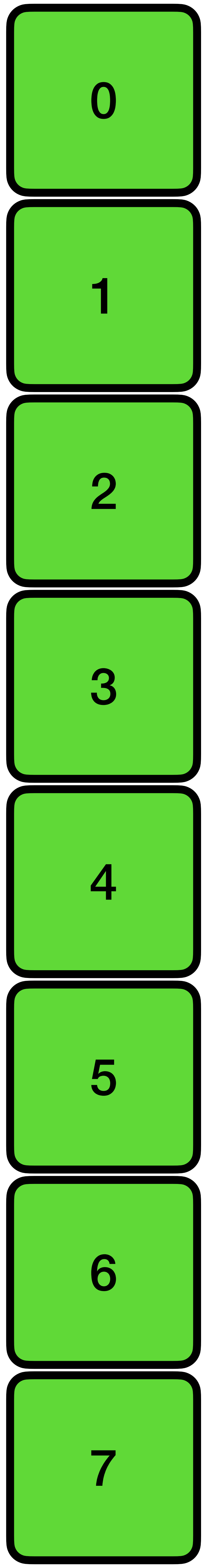


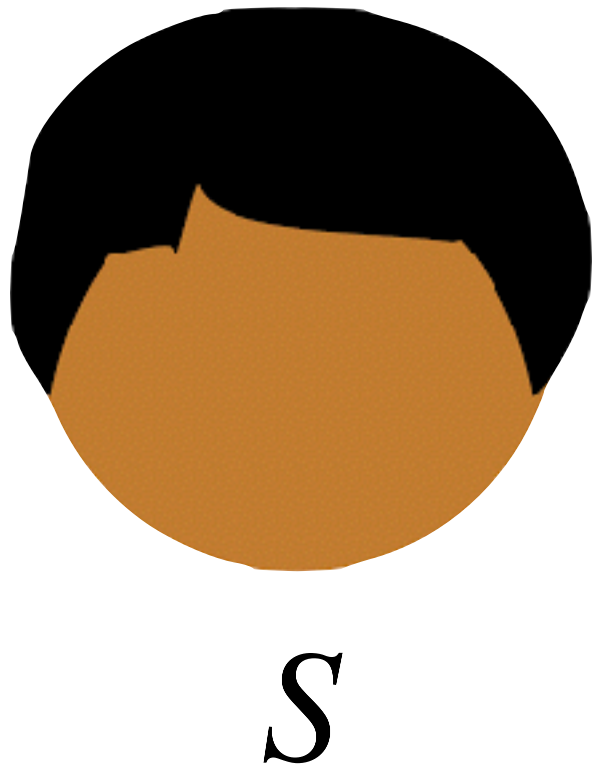


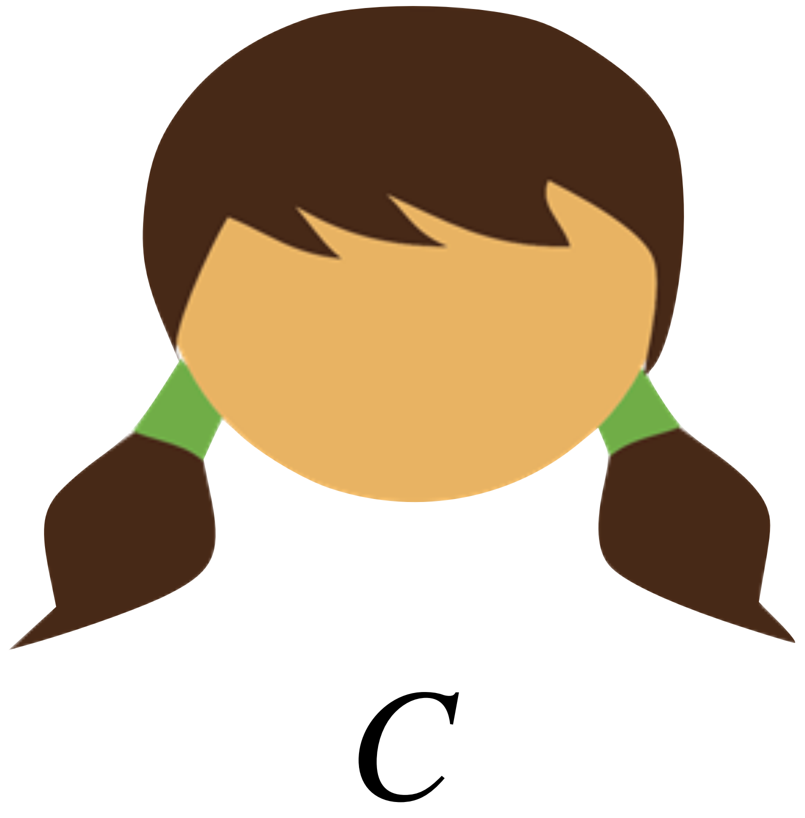
Basic idea: For each **logical** access, the client asks for multiple **physical** elements from the server

ORAM Security:

For a sequence of requests $\mathscr{R}$ from the client, the view of the server can be simulated







Step 1:
Encrypt RAM content

Enc(k, 0)

Enc(k, 1)

Enc(k, 2)

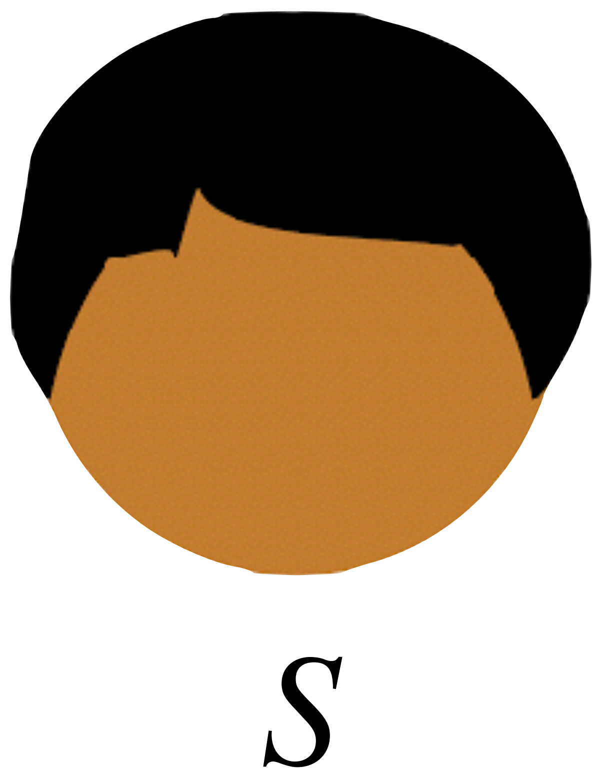
Enc(k, 3)

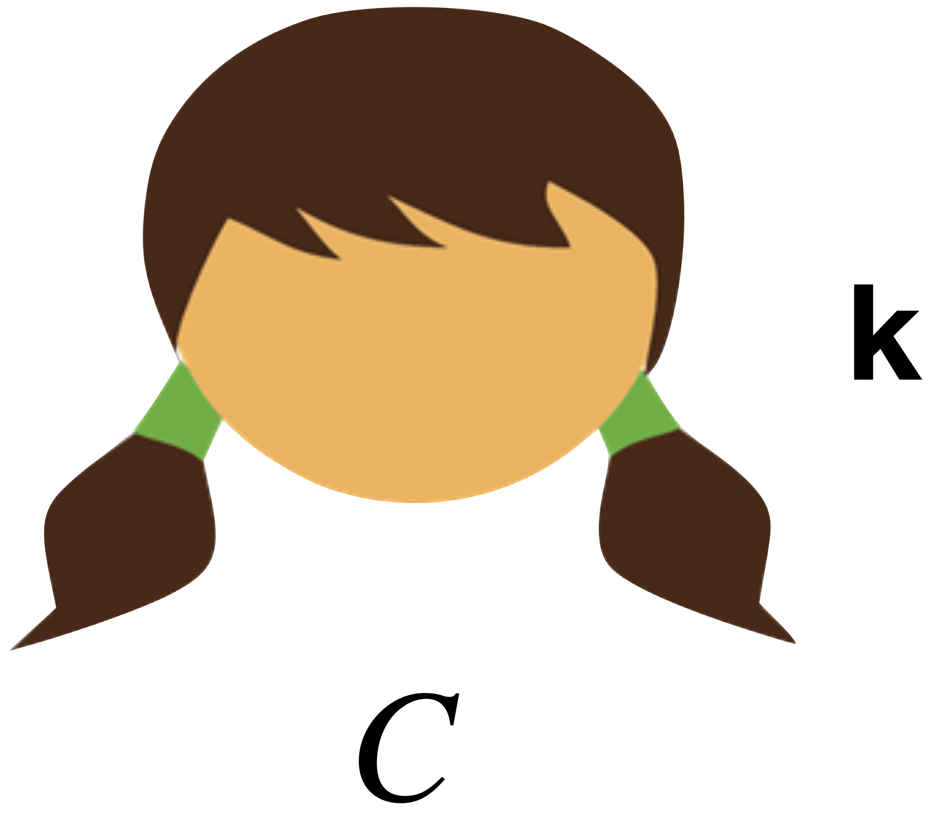
Enc(k, 4)

Enc(k, 5)

Enc(k, 6)

Enc(k, 7)





Step 1:
Encrypt RAM content

Enc(k, 0)

Enc(k, 1)

Enc(k, 2)

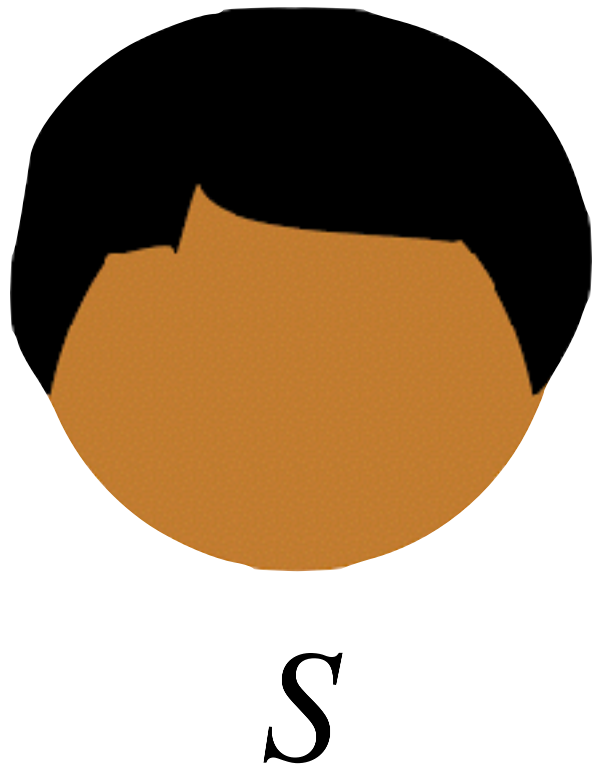
Enc(k, 3)

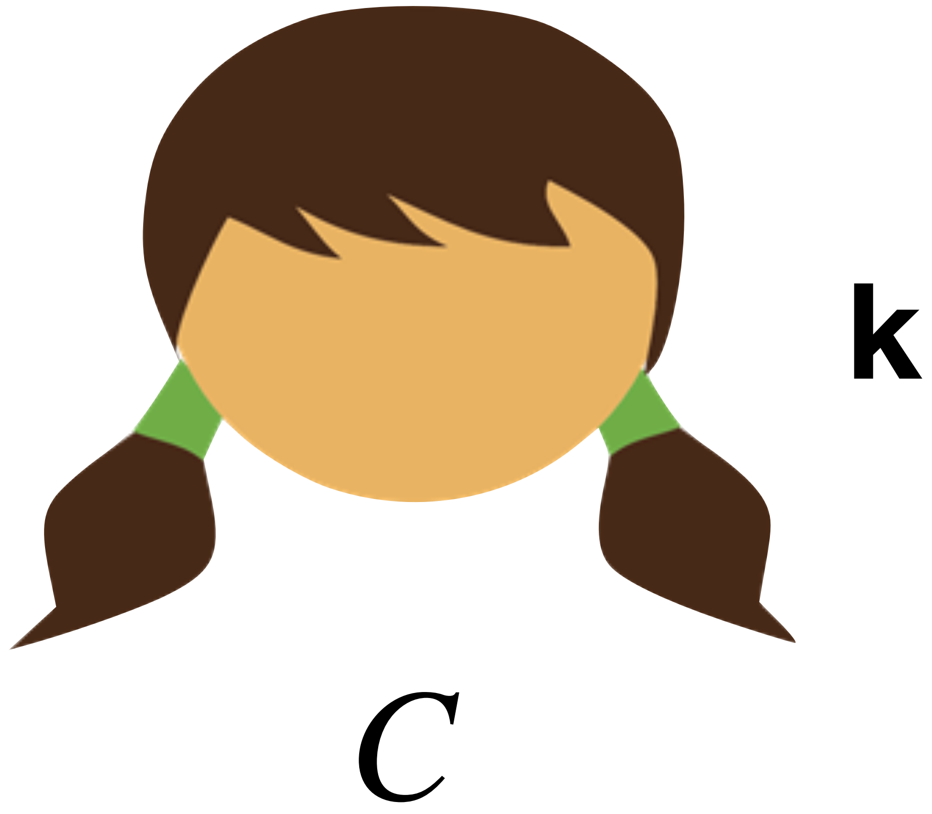
Enc(k, 4)

Enc(k, 5)

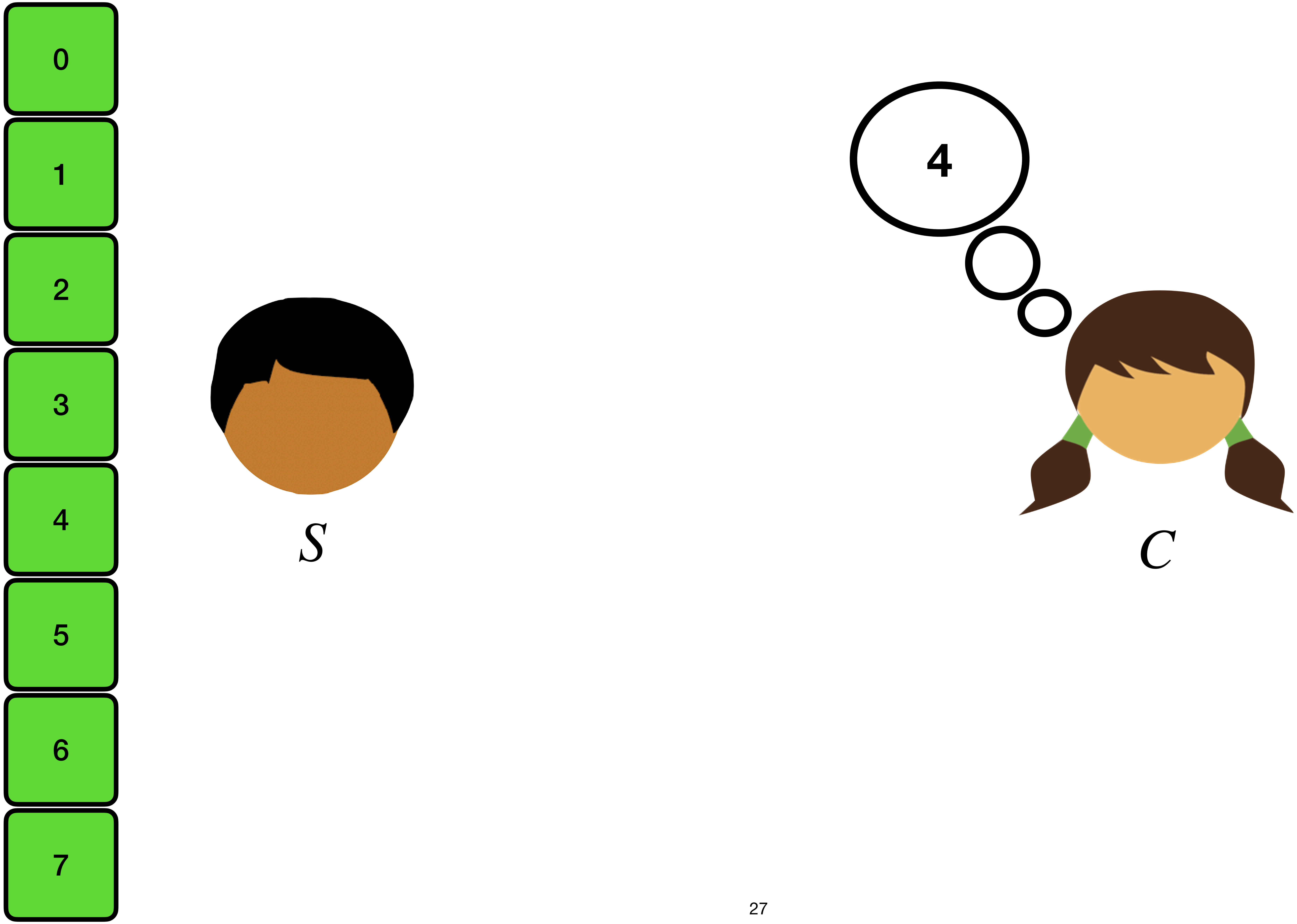
Enc(k, 6)

Enc(k, 7)

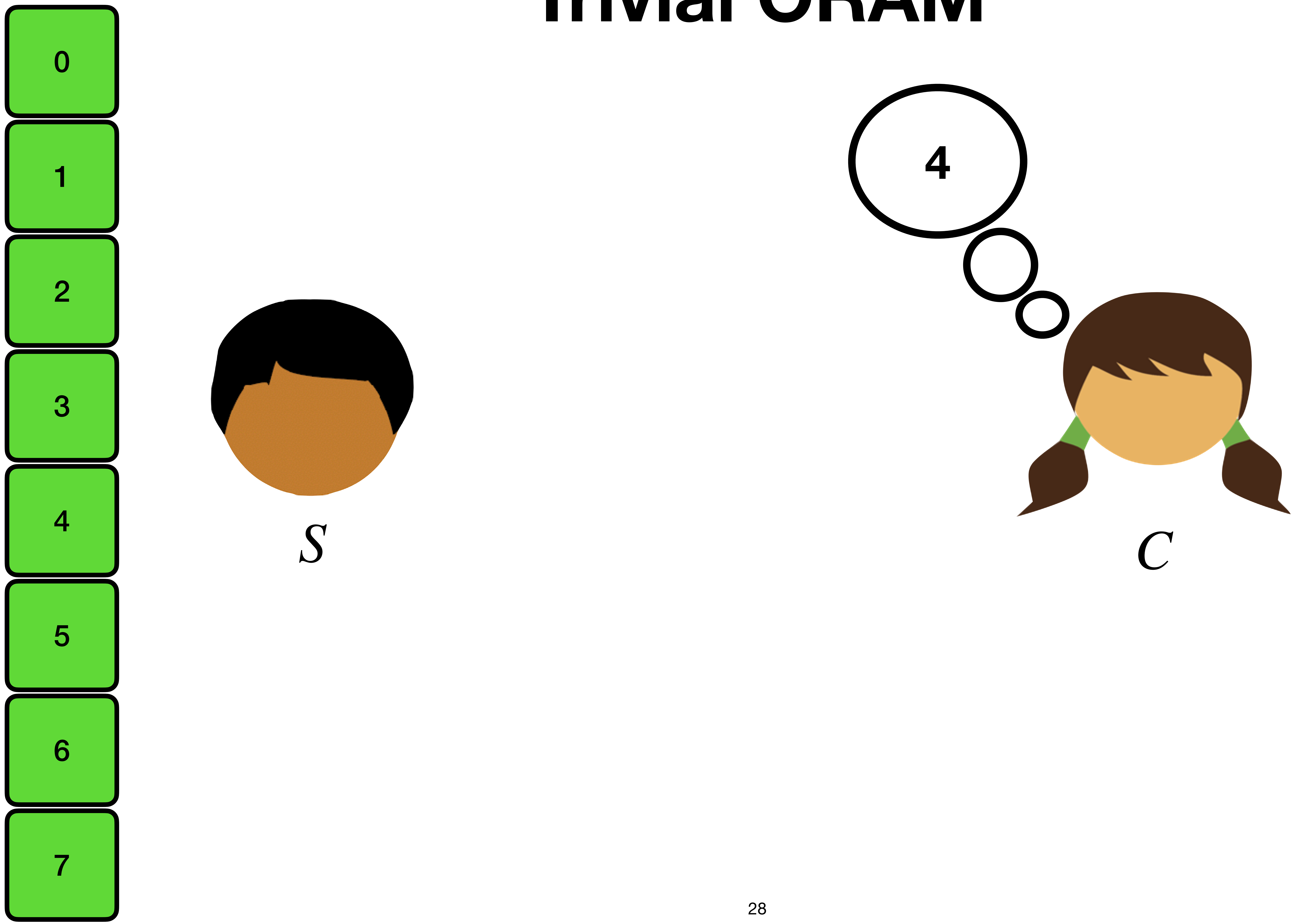




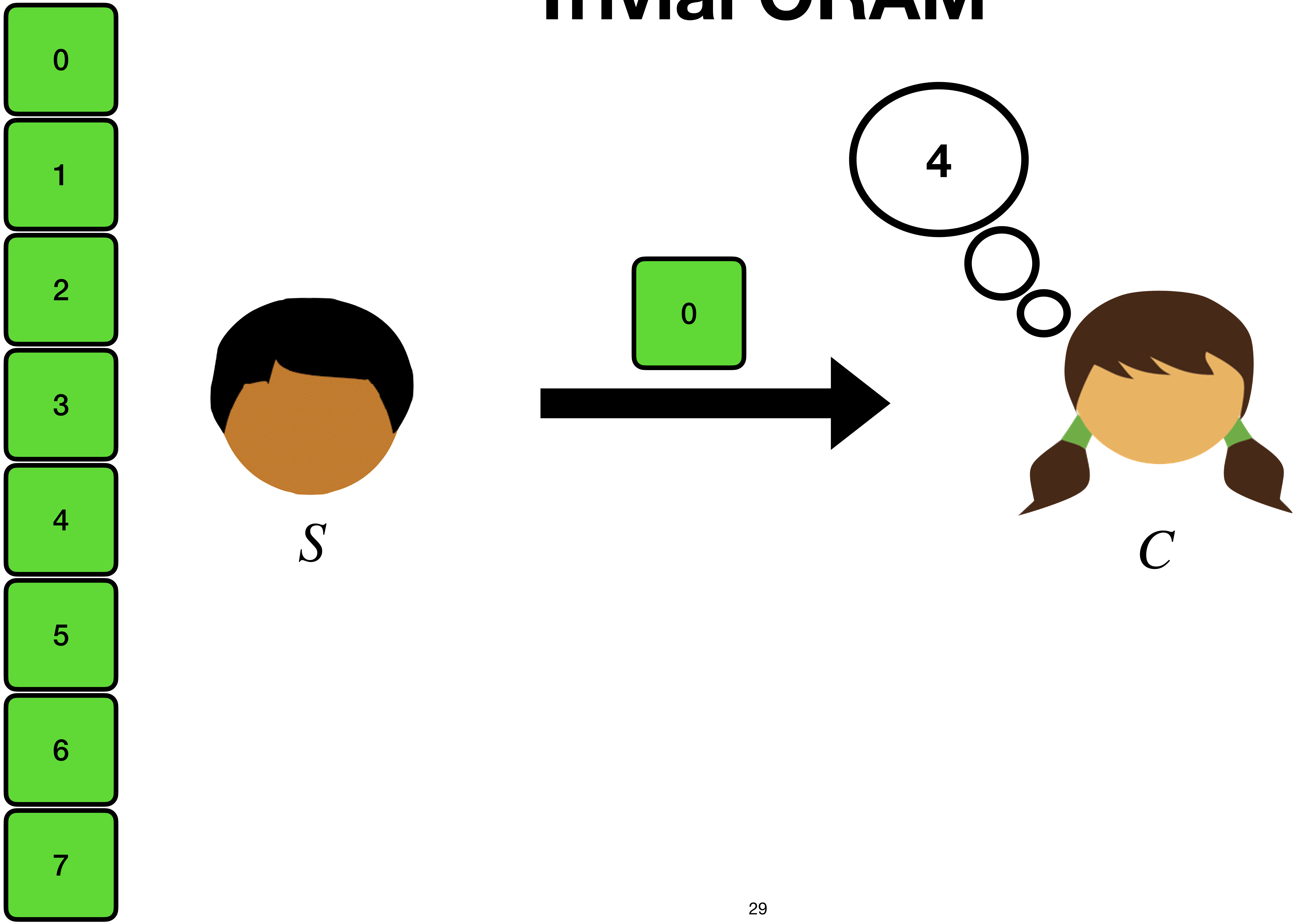
We will omit this from now on.
Assume all elements are encrypted.


0
1
2
3
4
5
6
7
S
4
C

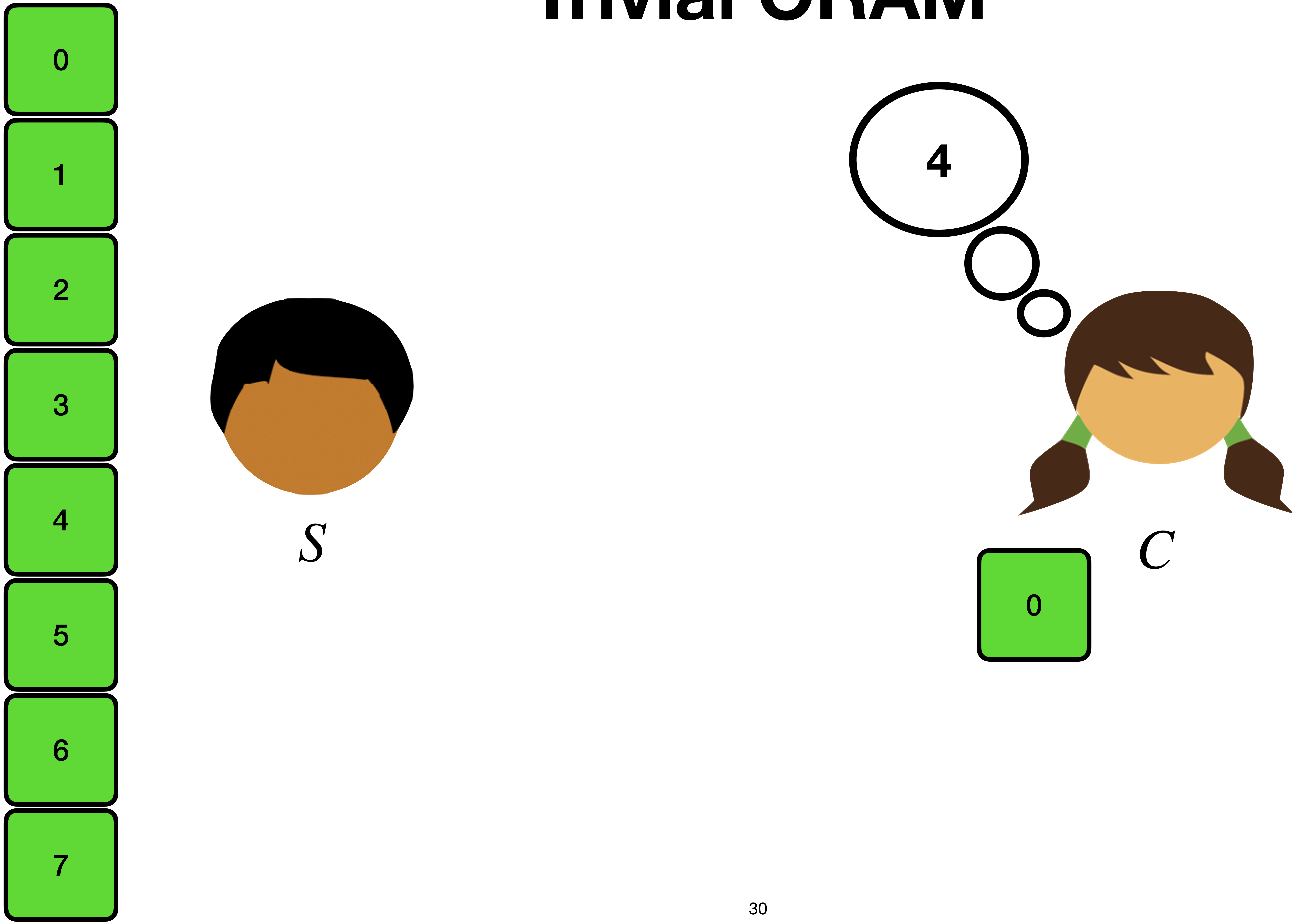
Trivial ORAM



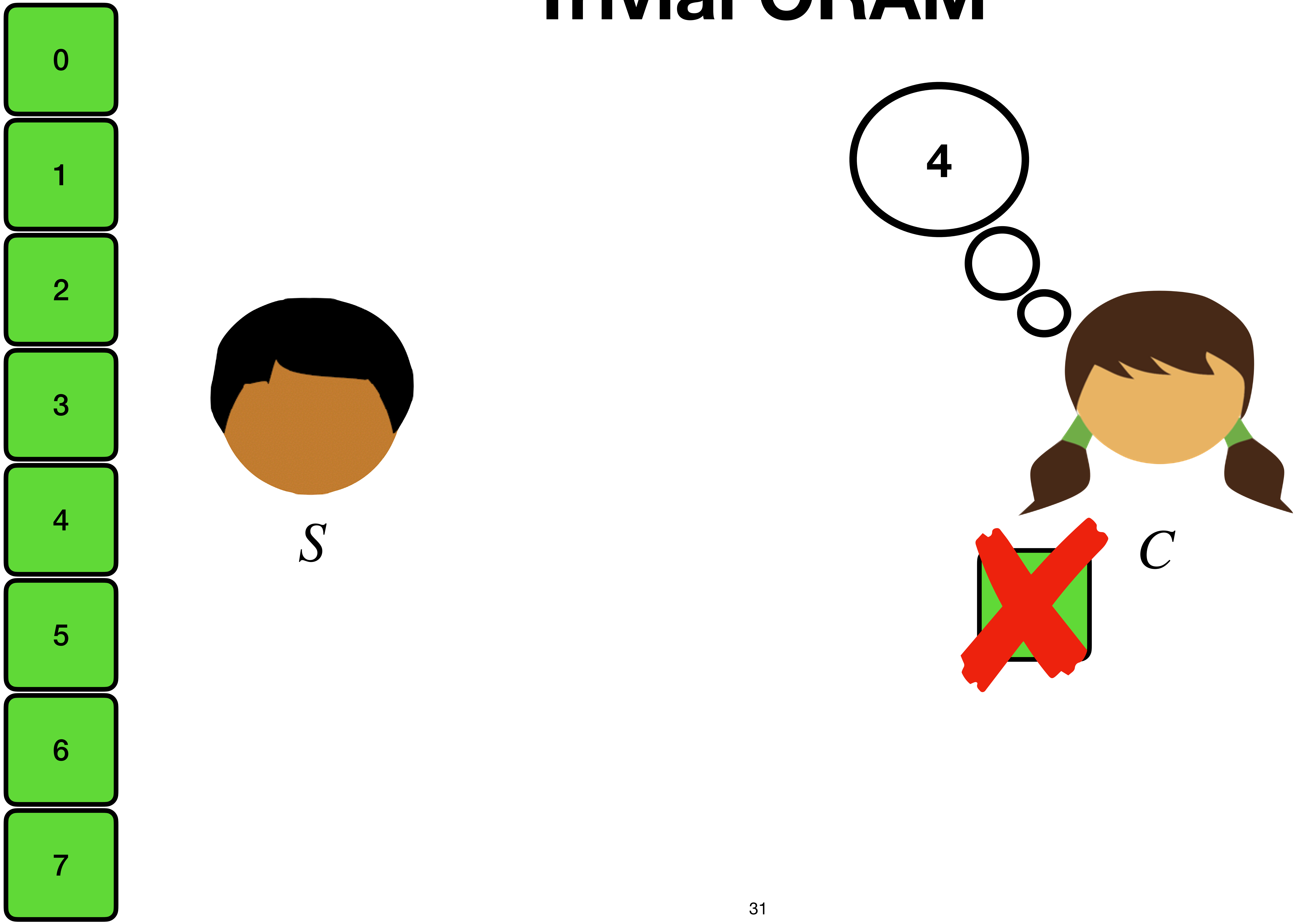
Trivial ORAM



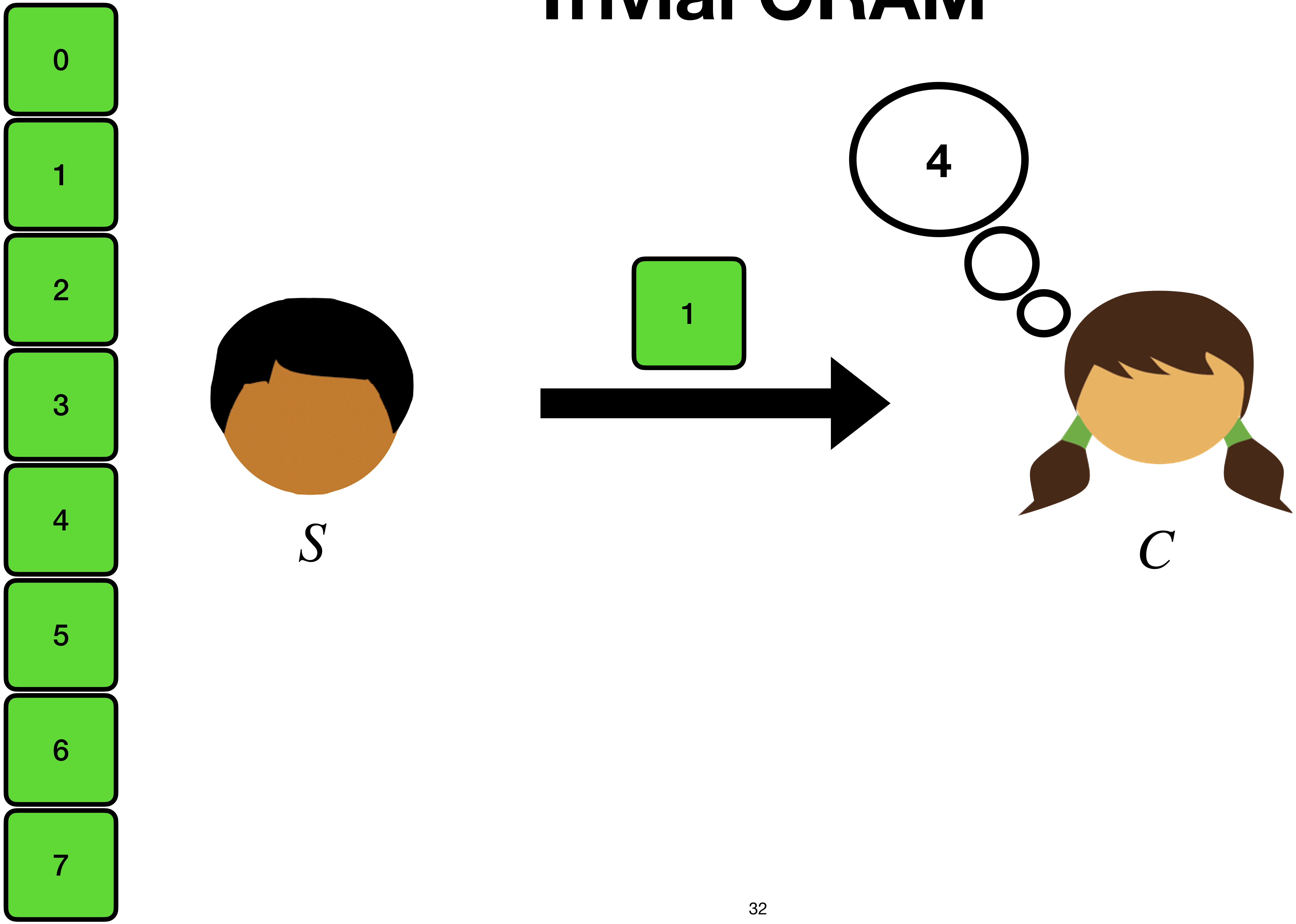
Trivial ORAM



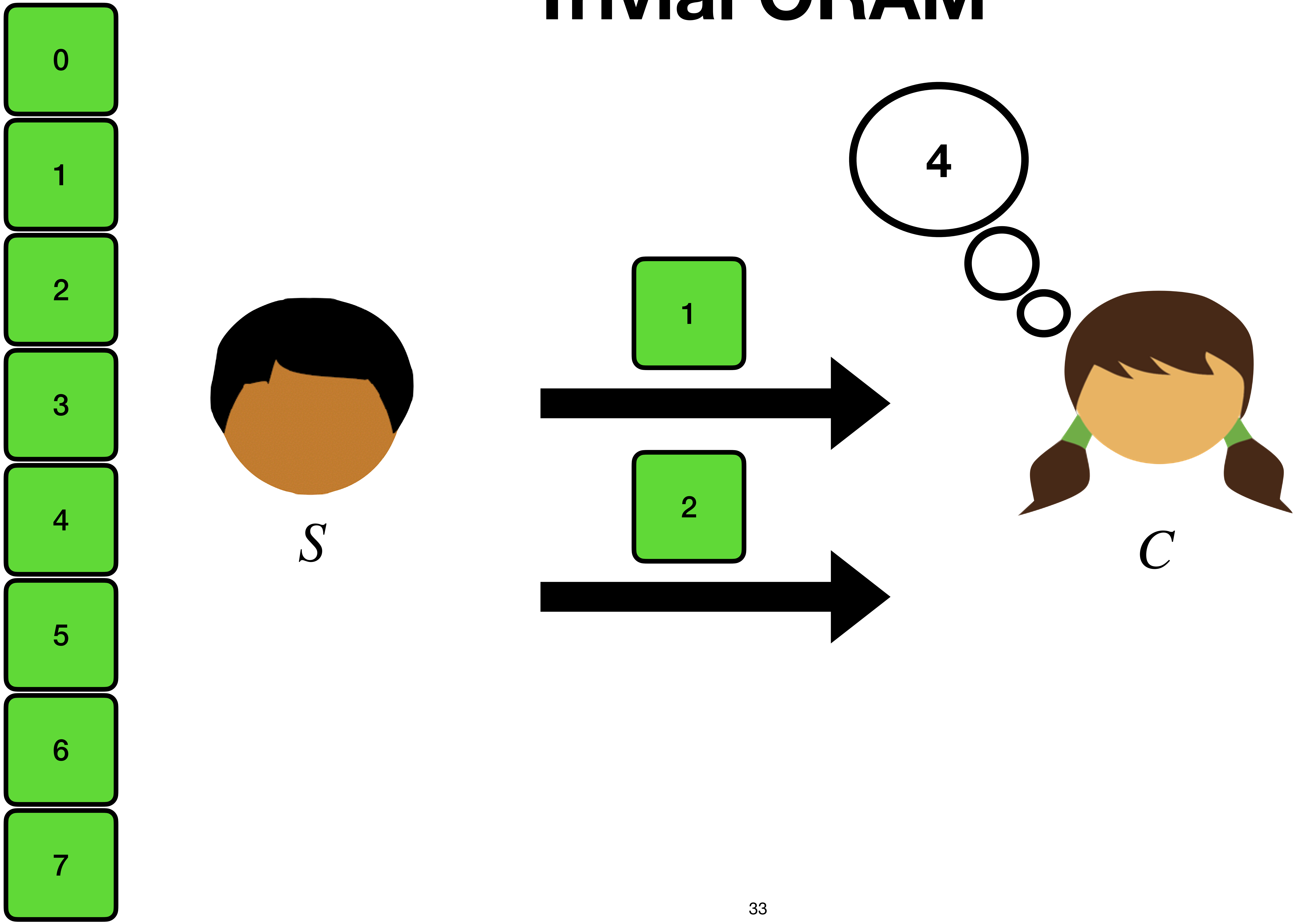
Trivial ORAM



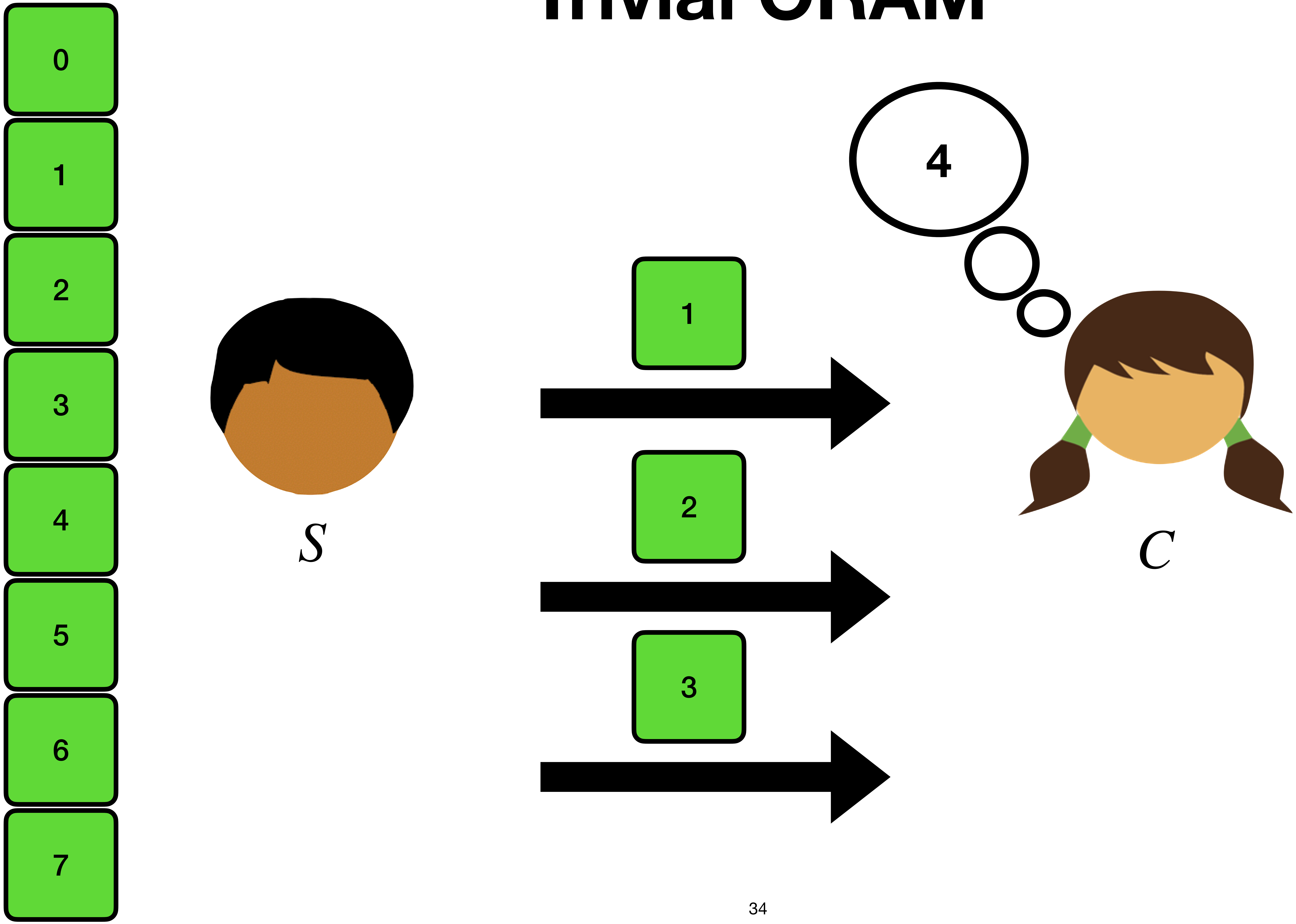
Trivial ORAM



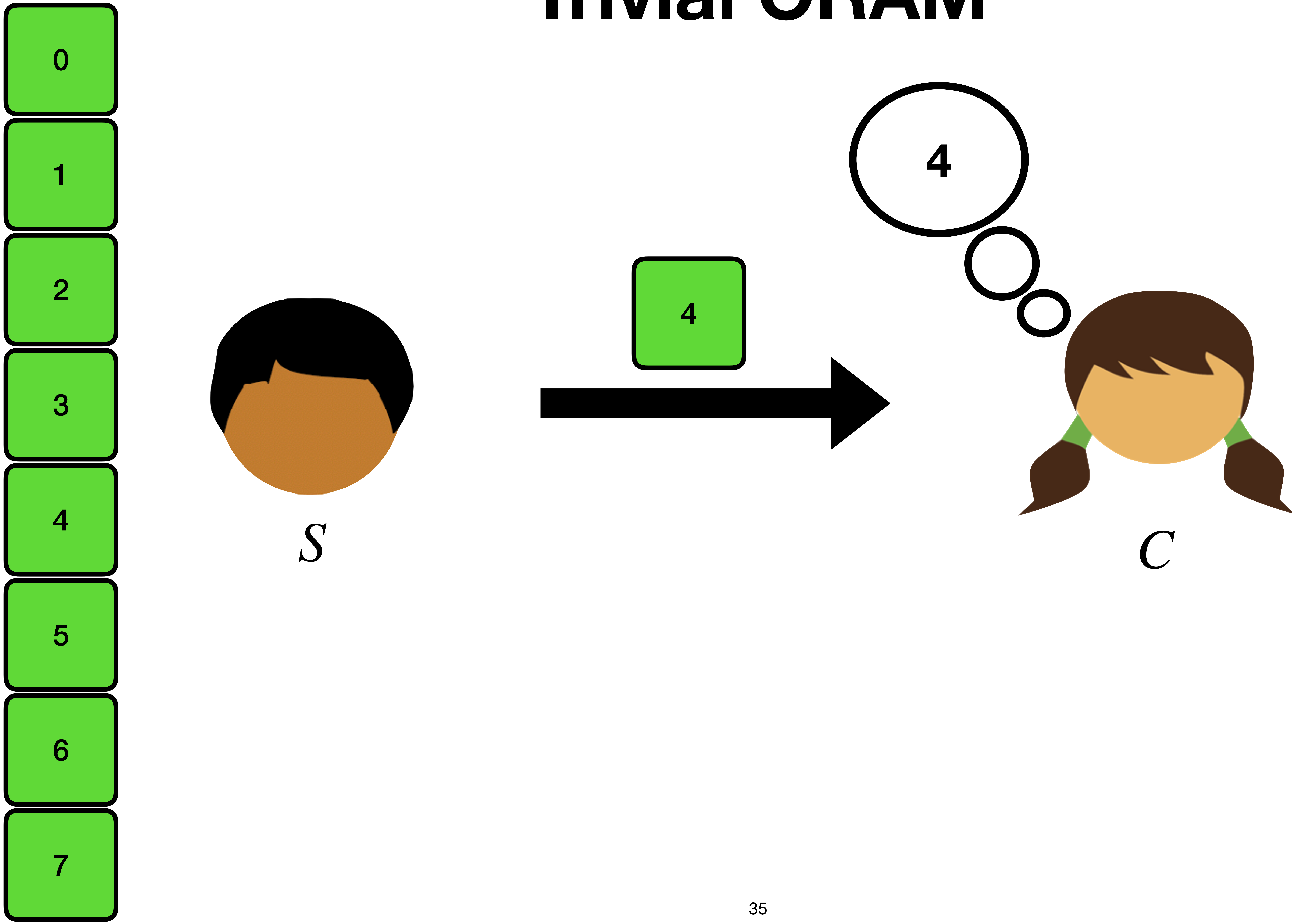
Trivial ORAM



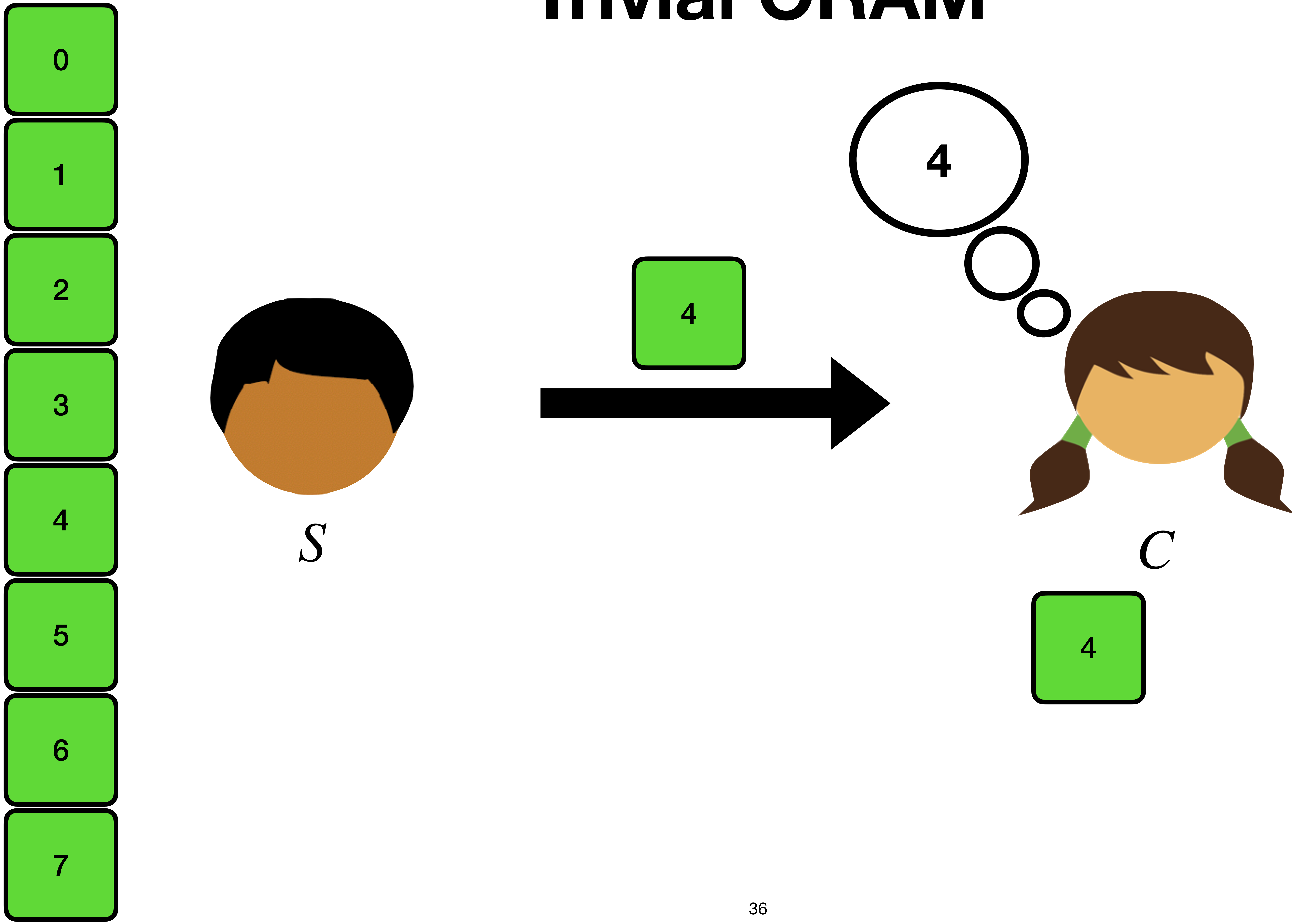
Trivial ORAM



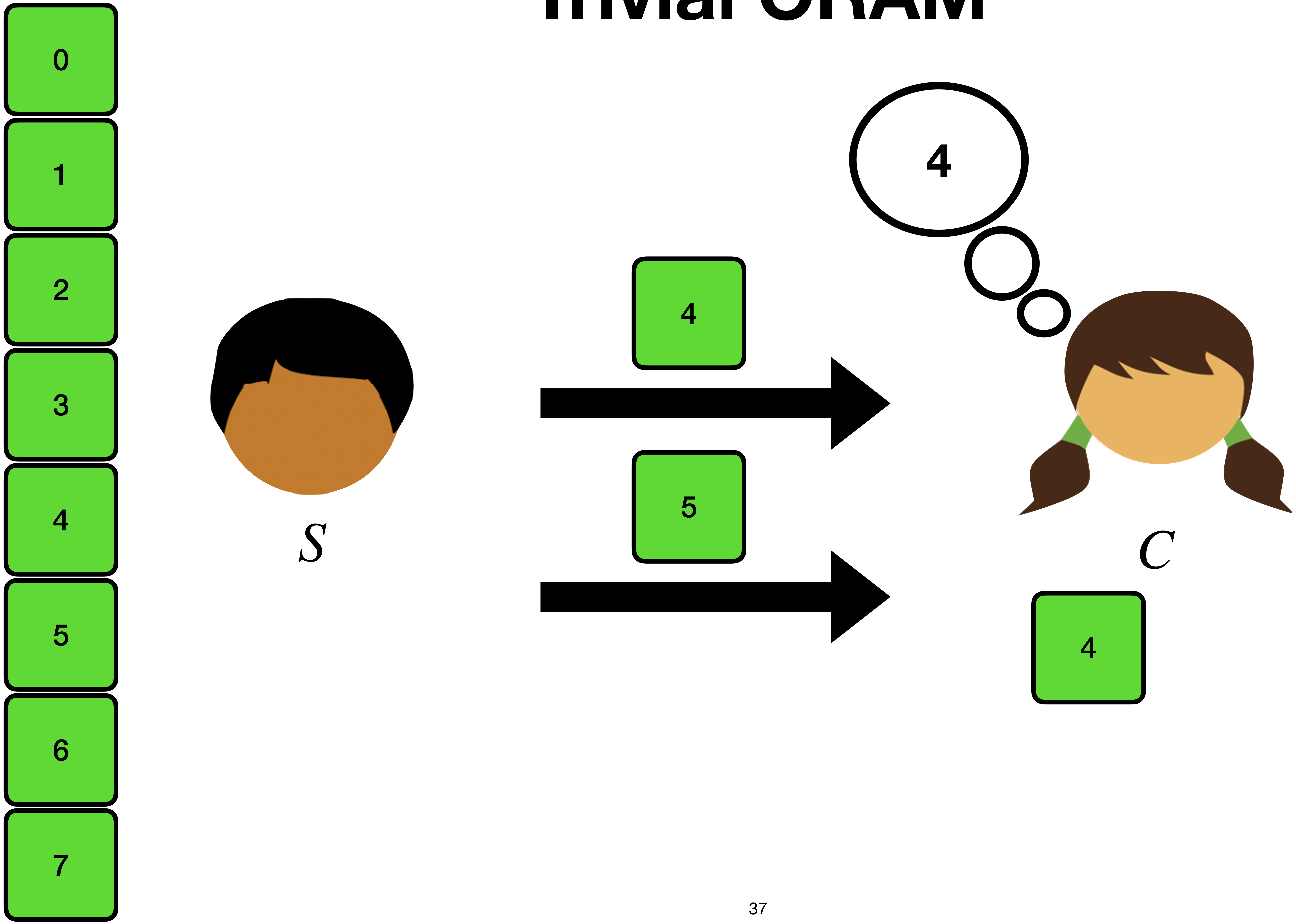
Trivial ORAM



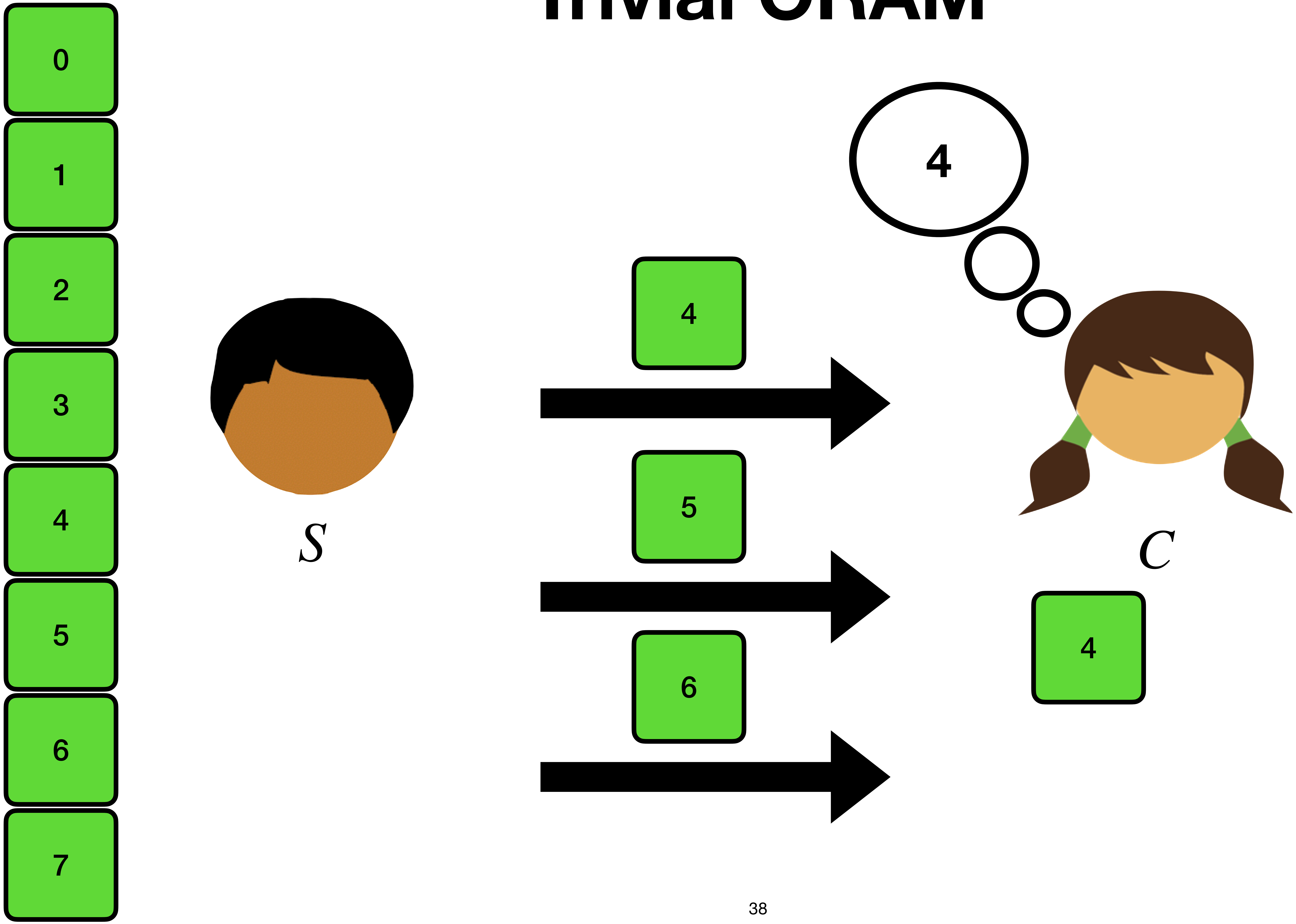
Trivial ORAM



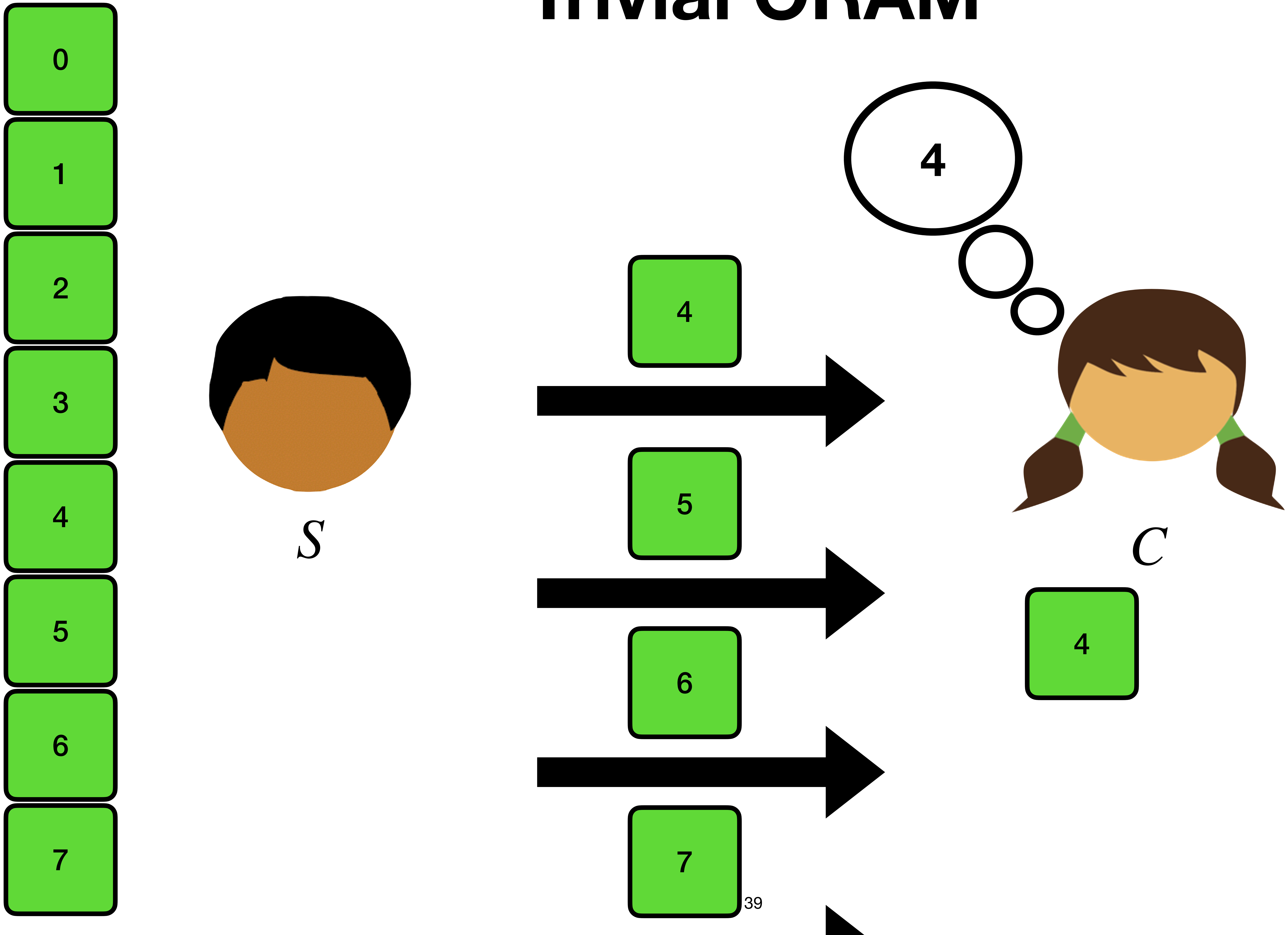
Trivial ORAM



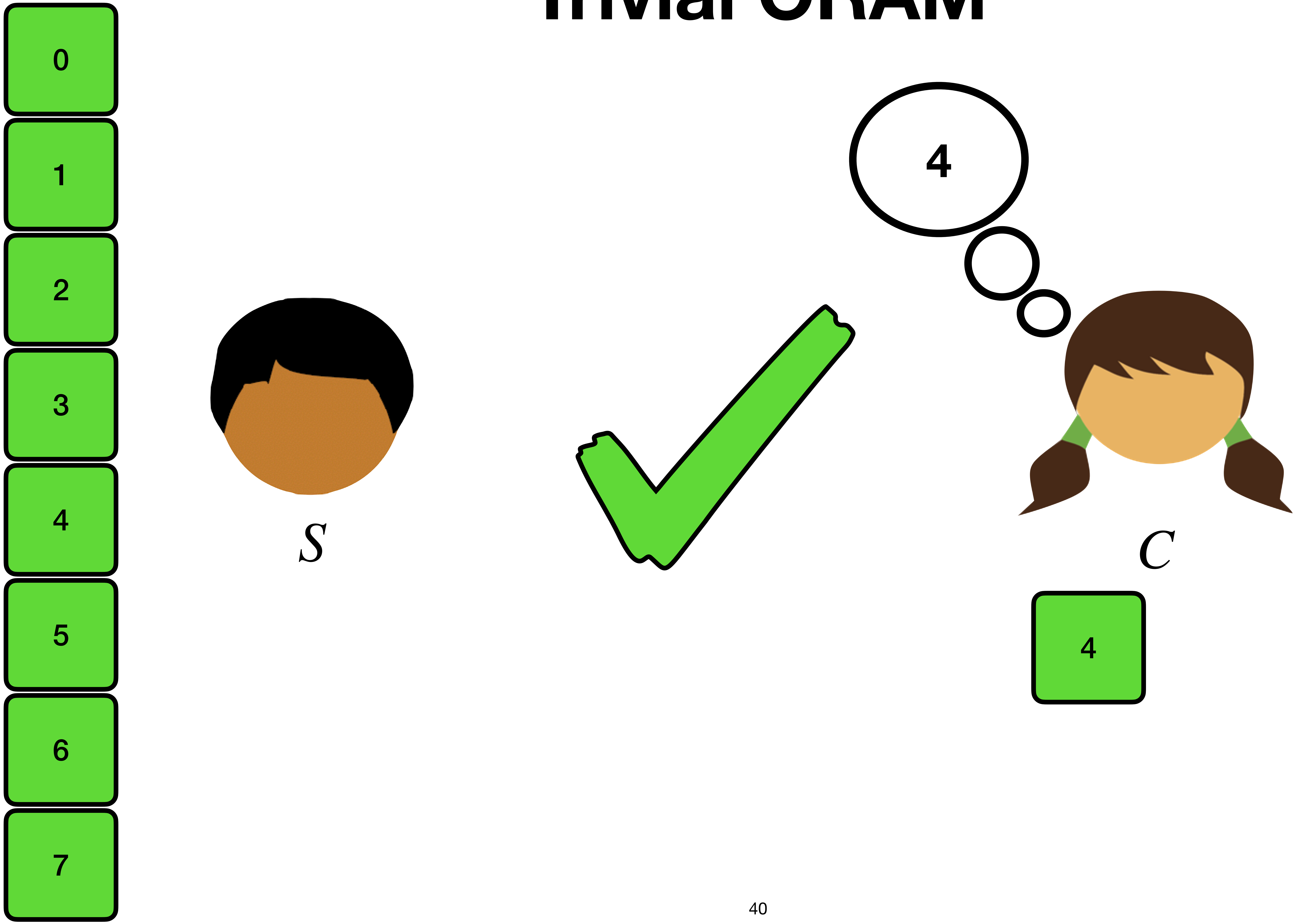
Trivial ORAM



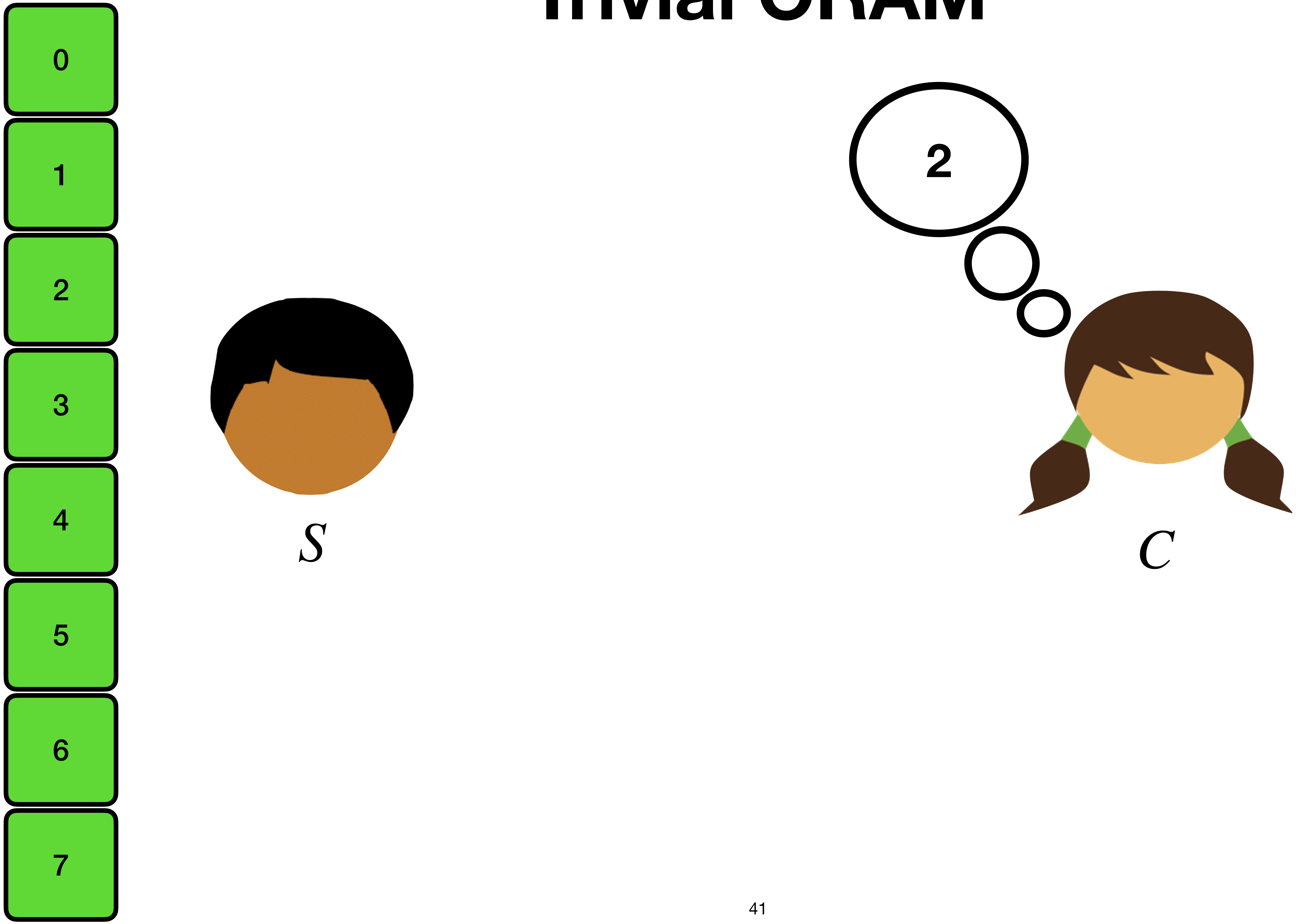
Trivial ORAM



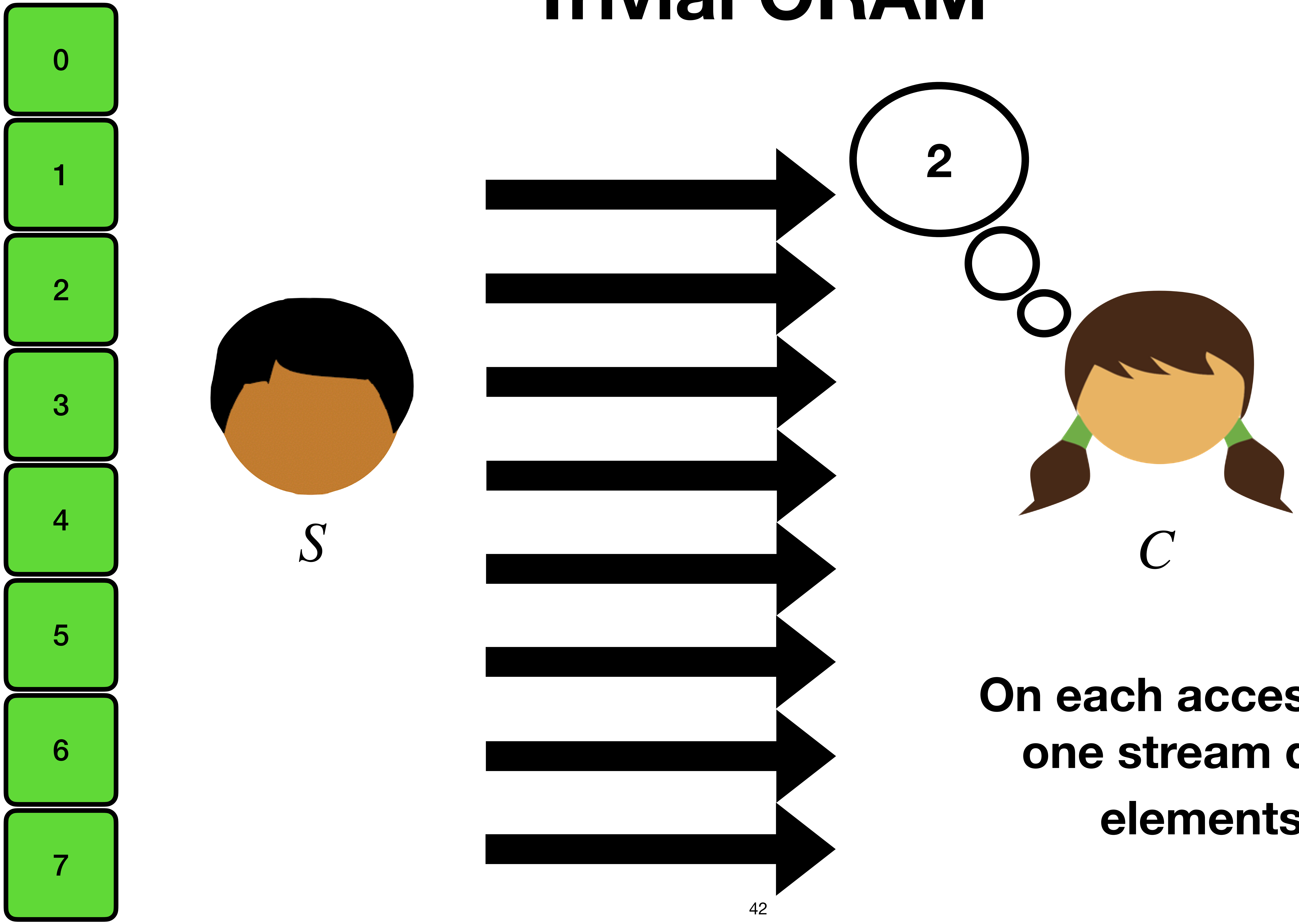
Trivial ORAM



Trivial ORAM

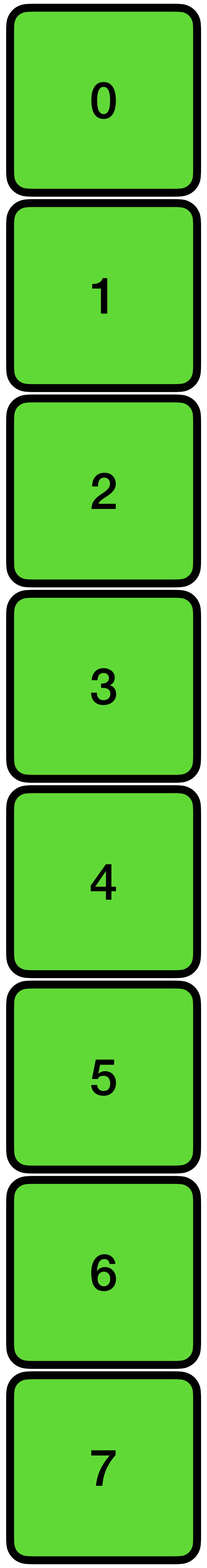


Trivial ORAM

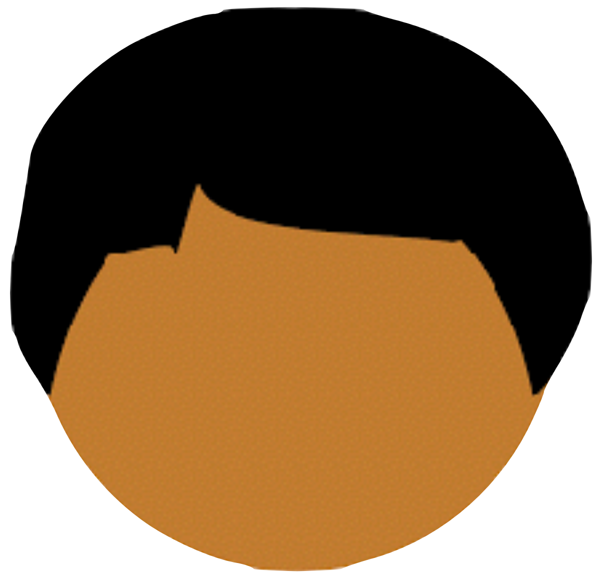


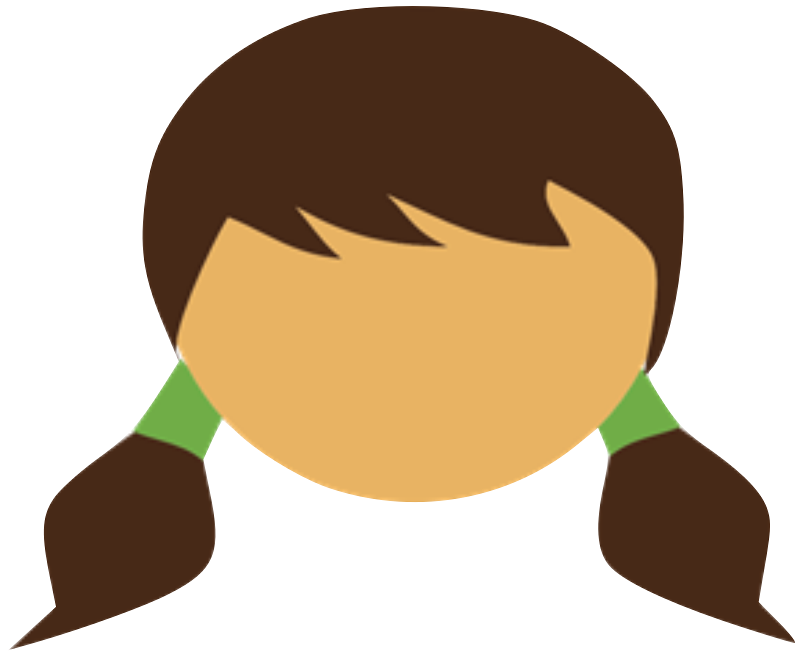
On each access, one-by-one stream database elements to C

Trivial ORAM



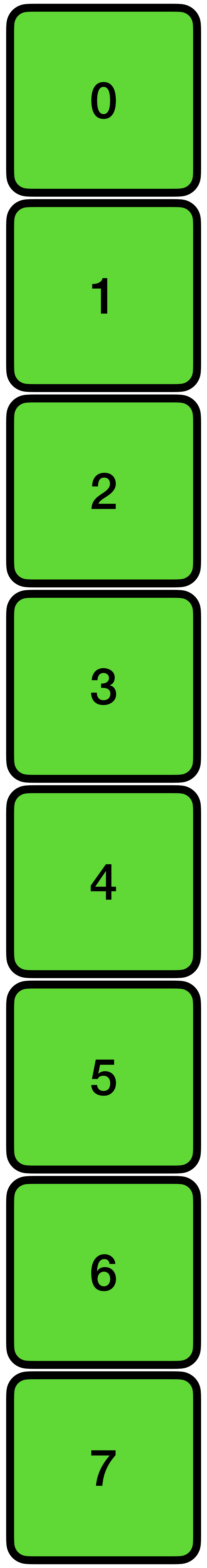
Client needs only $O(1)$ space ✓





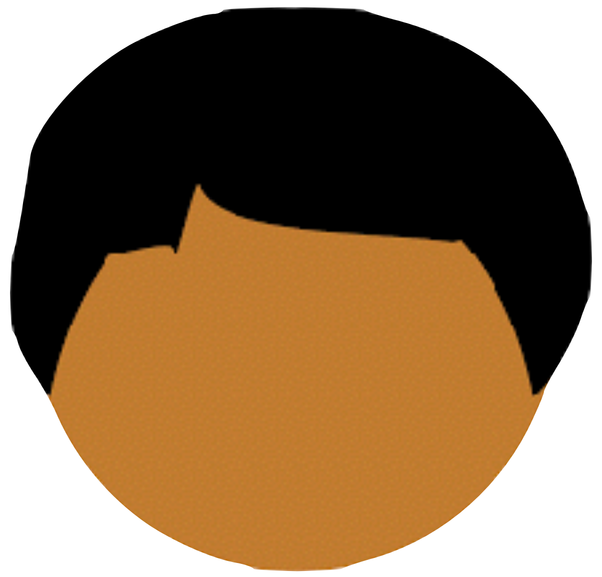
On each access, one-by-one stream database elements to C

Trivial ORAM

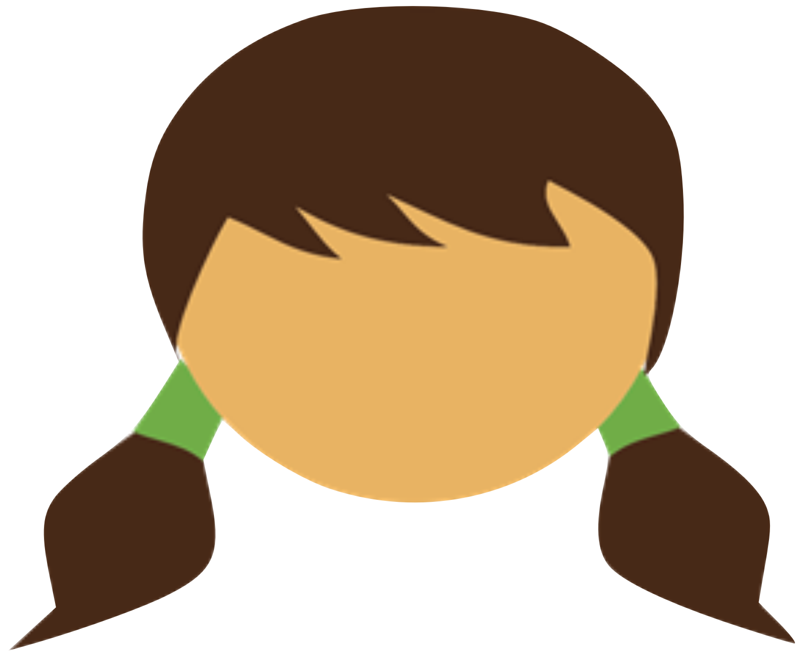


Client needs only $O(1)$ space ✓

Server is easy to simulate ✓



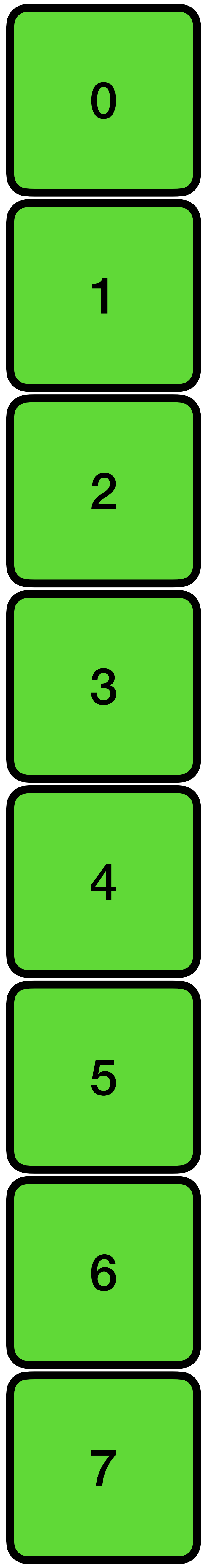
S

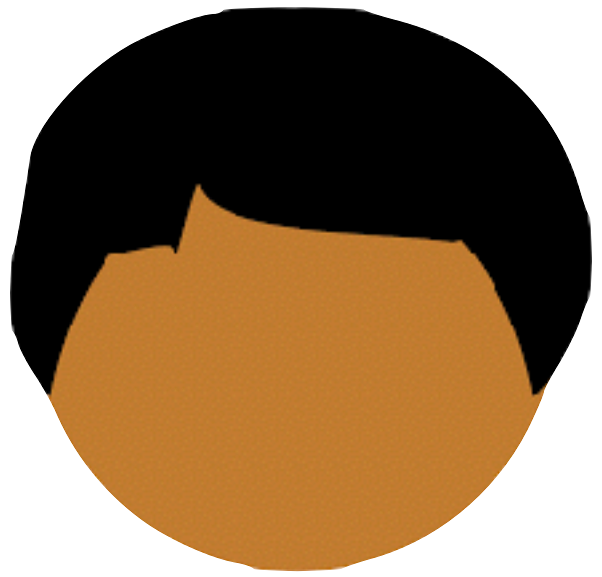


C

On each access, one-by-one stream database elements to C

Trivial ORAM



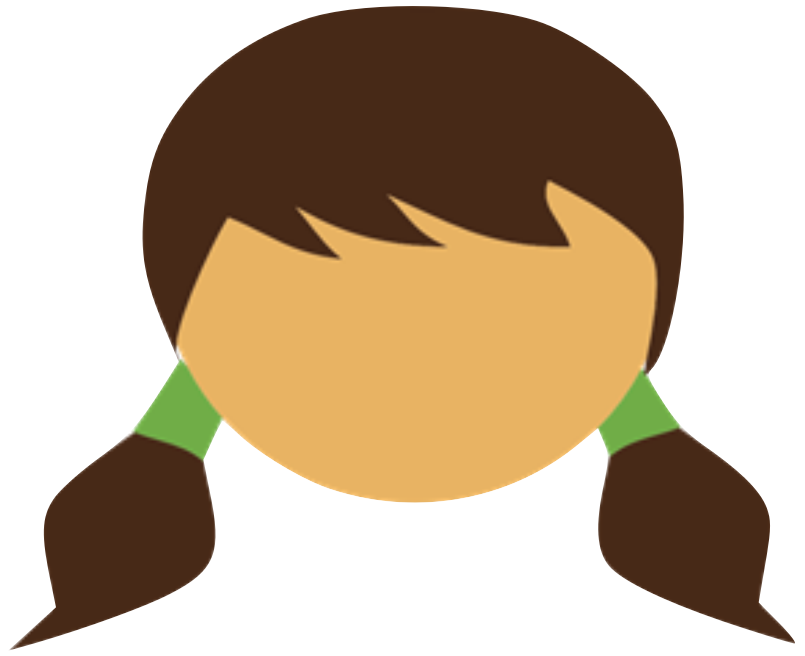


S

Client needs only $O(1)$ space ✓

Server is easy to simulate ✓

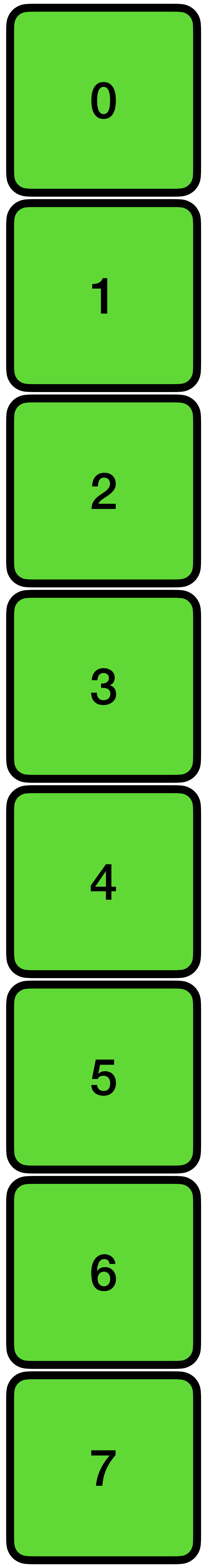
Linear overhead ✗

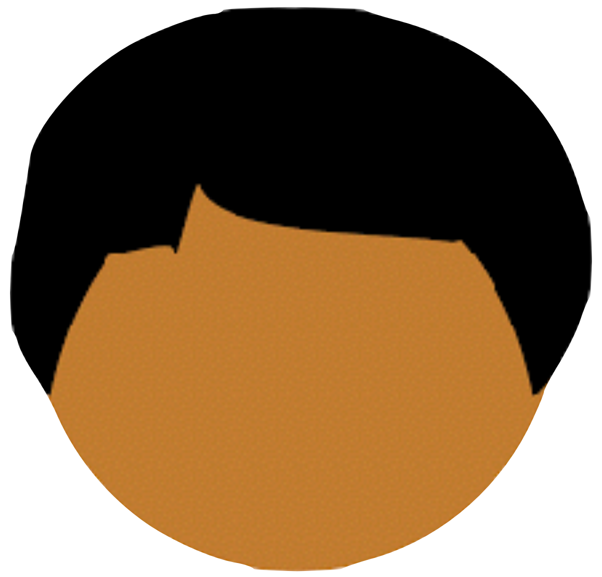


C

On each access, one-by-one stream database elements to C

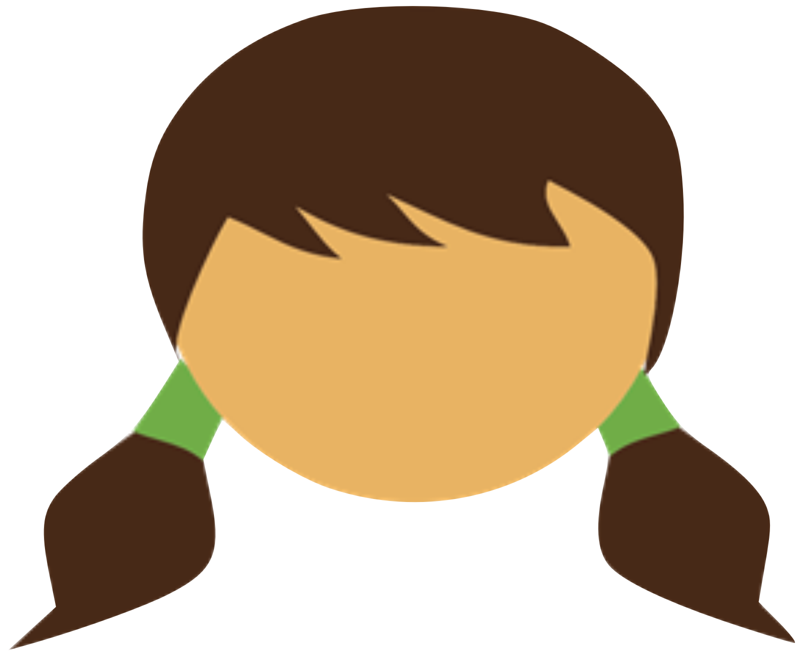
Non-Trivial ORAM





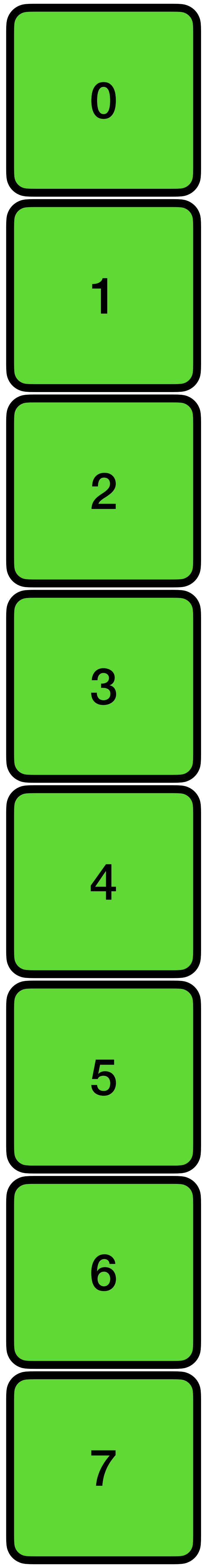
S

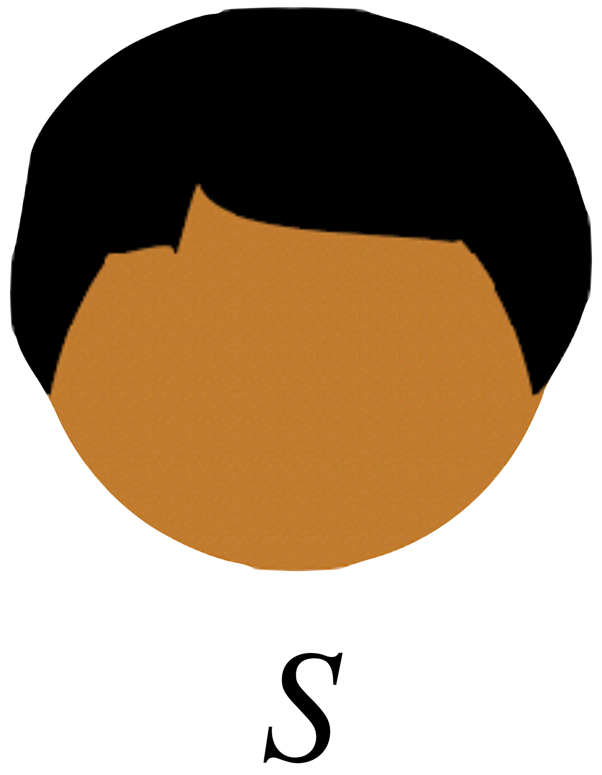
For each logical access, server needs to send only (amortized) $o(n)$ elements



C

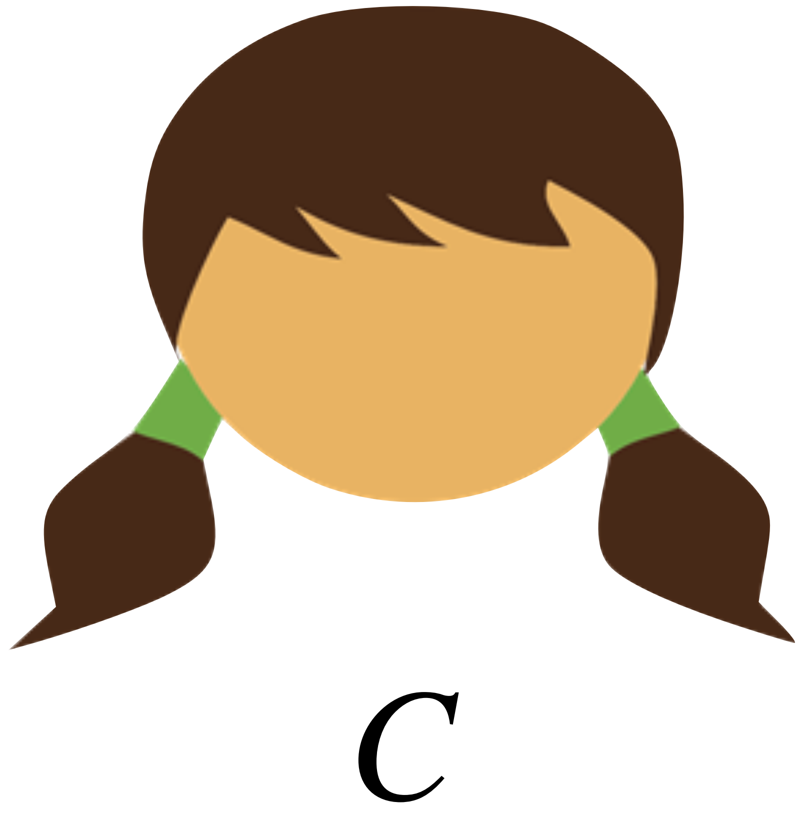
Non-Trivial ORAM



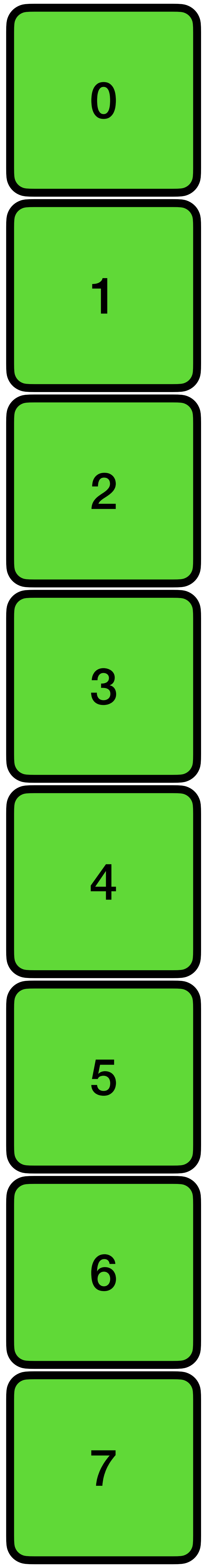


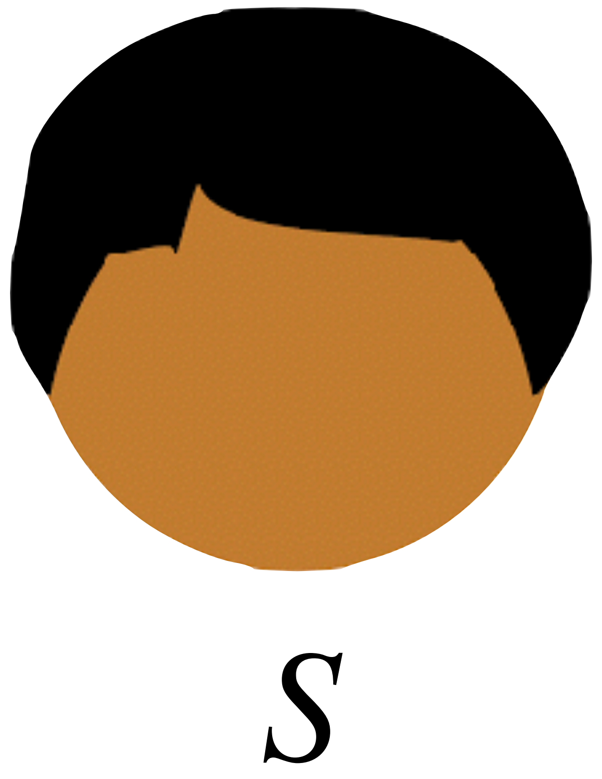
For each logical access, server needs to send only (amortized) $o(n)$ elements

Intuition: Client continually **shuffles** physical memory elements around



Non-Trivial ORAM

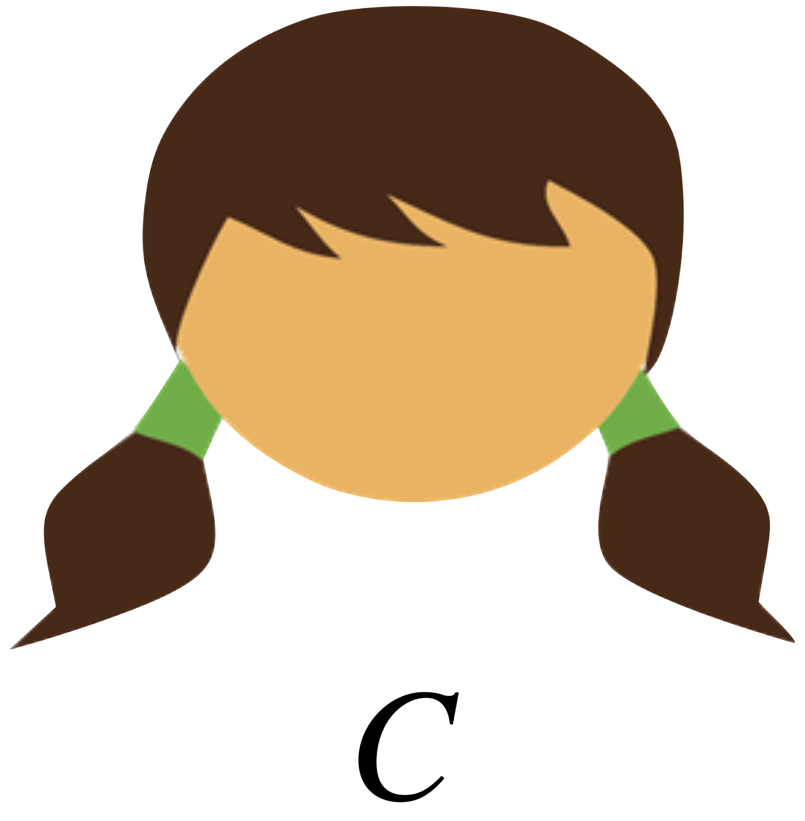


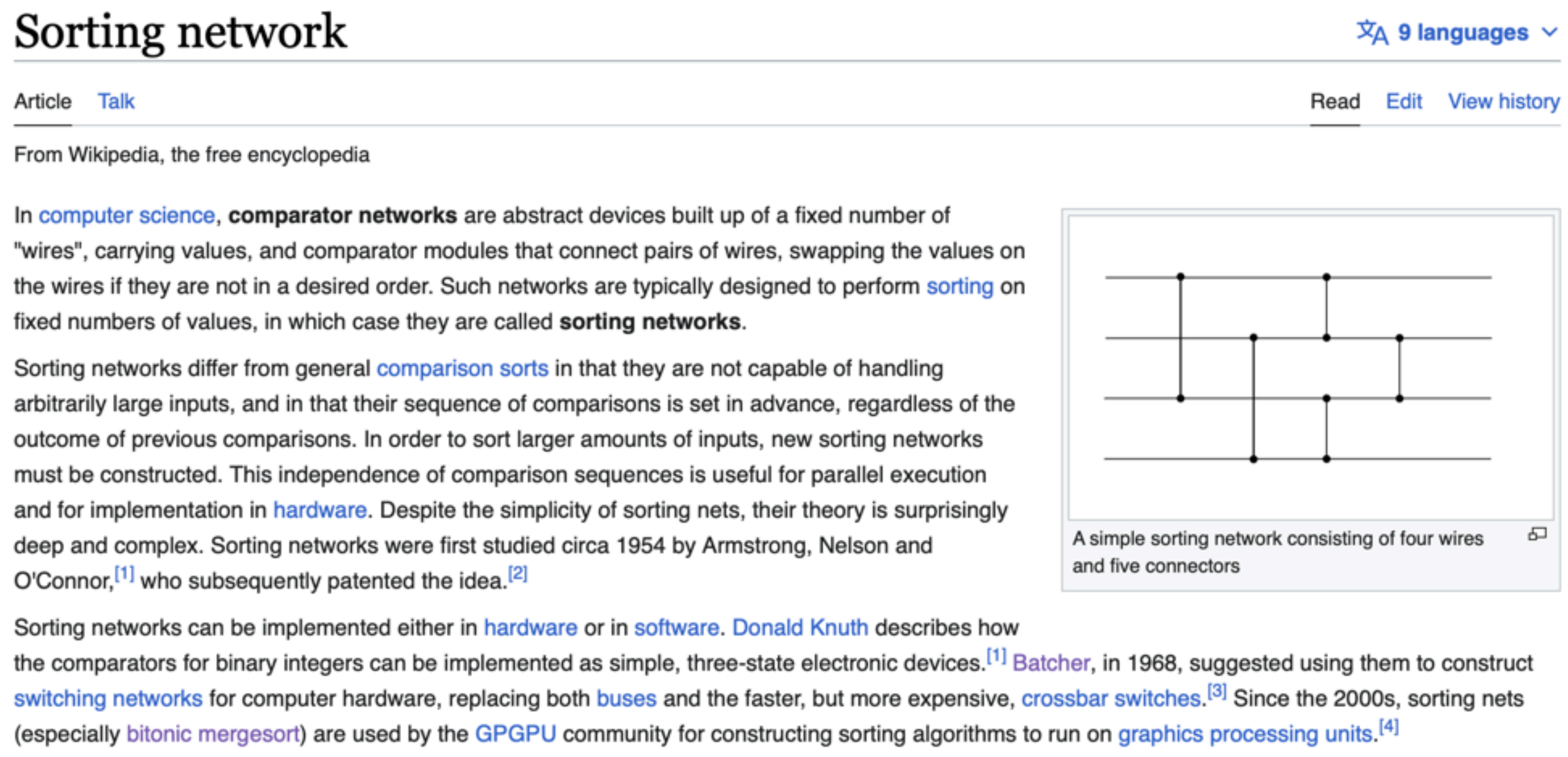


For each logical access, server needs to send only (amortized) $o(n)$ elements

Intuition: Client continually **shuffles** physical memory elements around

Problem: Client cannot store enough memory elements to shuffle them





Sorting network

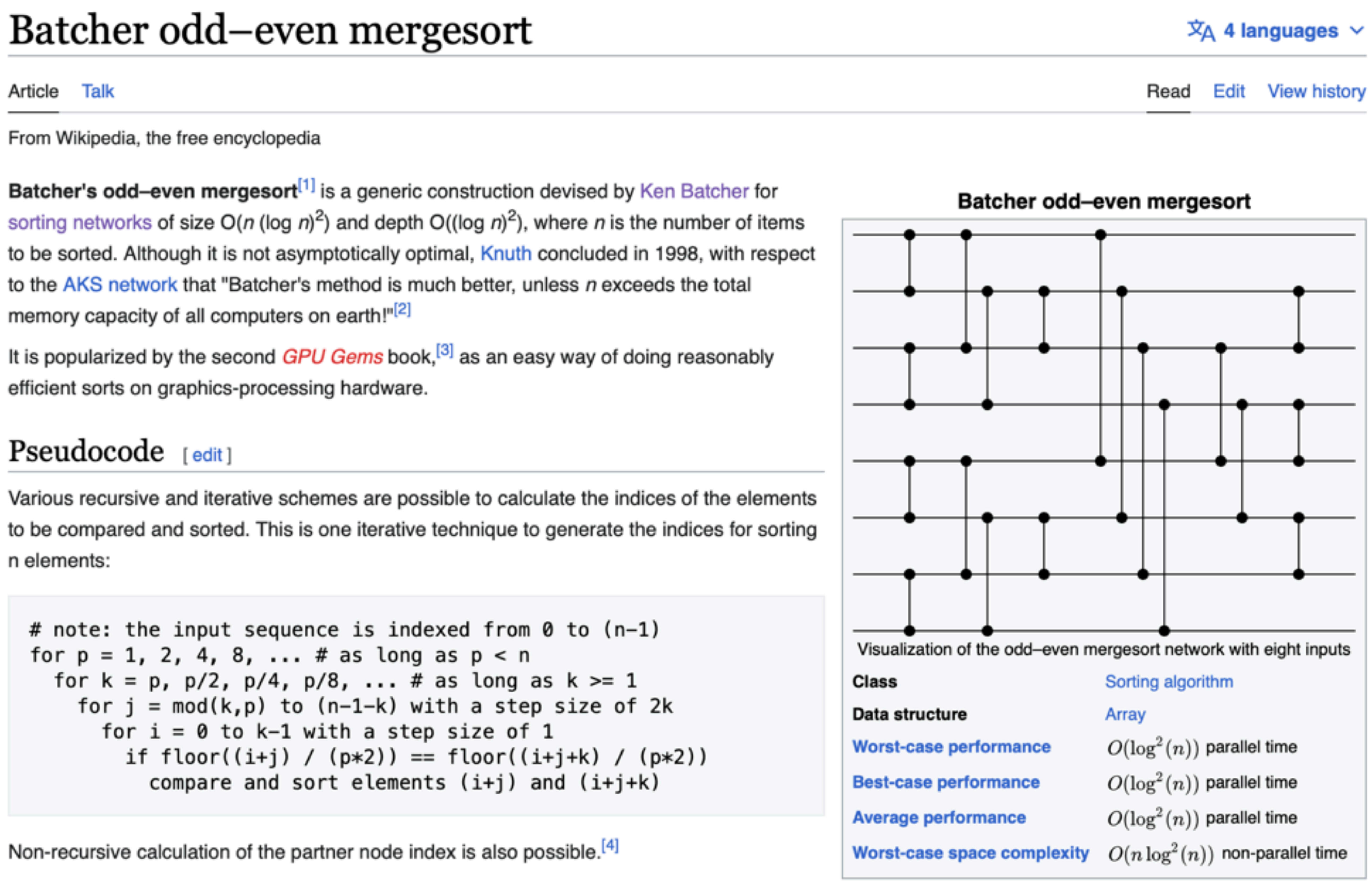
From Wikipedia, the free encyclopedia

In computer science, **comparator networks** are abstract devices built up of a fixed number of "wires", carrying values, and comparator modules that connect pairs of wires, swapping the values on the wires if they are not in a desired order. Such networks are typically designed to perform sorting on fixed numbers of values, in which case they are called **sorting networks**.

Sorting networks differ from general comparison sorts in that they are not capable of handling arbitrarily large inputs, and in that their sequence of comparisons is set in advance, regardless of the outcome of previous comparisons. In order to sort larger amounts of inputs, new sorting networks must be constructed. This independence of comparison sequences is useful for parallel execution and for implementation in hardware. Despite the simplicity of sorting nets, their theory is surprisingly deep and complex. Sorting networks were first studied circa 1954 by Armstrong, Nelson and O'Connor,[1] who subsequently patented the idea.[2]

A simple sorting network consisting of four wires and five connectors

Sorting networks can be implemented either in hardware or in software. Donald Knuth describes how the comparators for binary integers can be implemented as simple, three-state electronic devices.[1] Batcher, in 1968, suggested using them to construct switching networks for computer hardware, replacing both buses and the faster, but more expensive, crossbar switches.[3] Since the 2000s, sorting nets (especially bitonic mergesort) are used by the GPGPU community for constructing sorting algorithms to run on graphics processing units.[4]



Batcher odd–even mergesort

文A 4 languages ⌄

Article Talk

Read Edit View history

From Wikipedia, the free encyclopedia

Batcher's odd–even mergesort[1] is a generic construction devised by Ken Batcher for sorting networks of size O(*n* (log *n*)^2) and depth O((log *n*)^2), where *n* is the number of items to be sorted. Although it is not asymptotically optimal, Knuth concluded in 1998, with respect to the AKS network that "Batcher's method is much better, unless *n* exceeds the total memory capacity of all computers on earth!"[2]

It is popularized by the second *GPU Gems* book,[3] as an easy way of doing reasonably efficient sorts on graphics-processing hardware.

Batcher odd–even mergesort

Visualization of the odd–even mergesort network with eight inputs

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(\log^2(n))$ parallel time
Best-case performance	$O(\log^2(n))$ parallel time
Average performance	$O(\log^2(n))$ parallel time
Worst-case space complexity	$O(n\log^2(n))$ non-parallel time

Pseudocode [edit]

Various recursive and iterative schemes are possible to calculate the indices of the elements to be compared and sorted. This is one iterative technique to generate the indices for sorting n elements:

```
# note: the input sequence is indexed from 0 to (n-1)
for p = 1, 2, 4, 8, ... # as long as p < n
  for k = p, p/2, p/4, p/8, ... # as long as k >= 1
    for j = mod(k,p) to (n-1-k) with a step size of 2k
      for i = 0 to k-1 with a step size of 1
        if floor((i+j) / (p*2)) == floor((i+j+k) / (p*2))
          compare and sort elements (i+j) and (i+j+k)
```

Non-recursive calculation of the partner node index is also possible.[4]

Pseudorandom Function (PRF)

A function family F is considered pseudorandom *if the following indistinguishability holds*

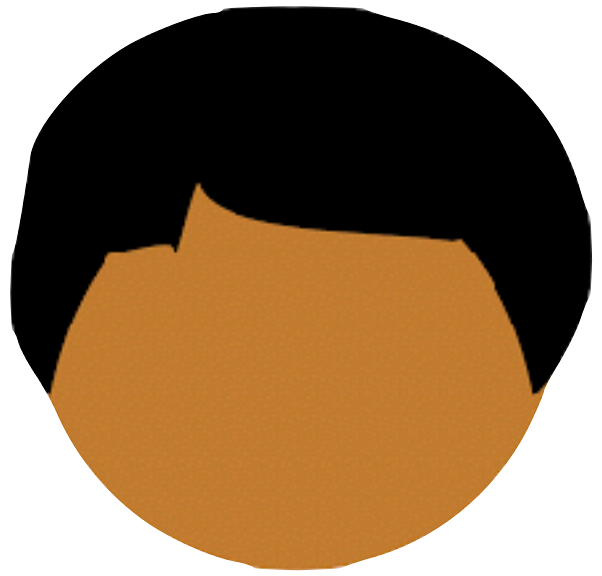
```
Real:
    $k \xleftarrow{\$} \{0,1\}^\lambda$

    lookup($m$):
        return $F(k, m)$
```

$$\overset{c}{=}$$

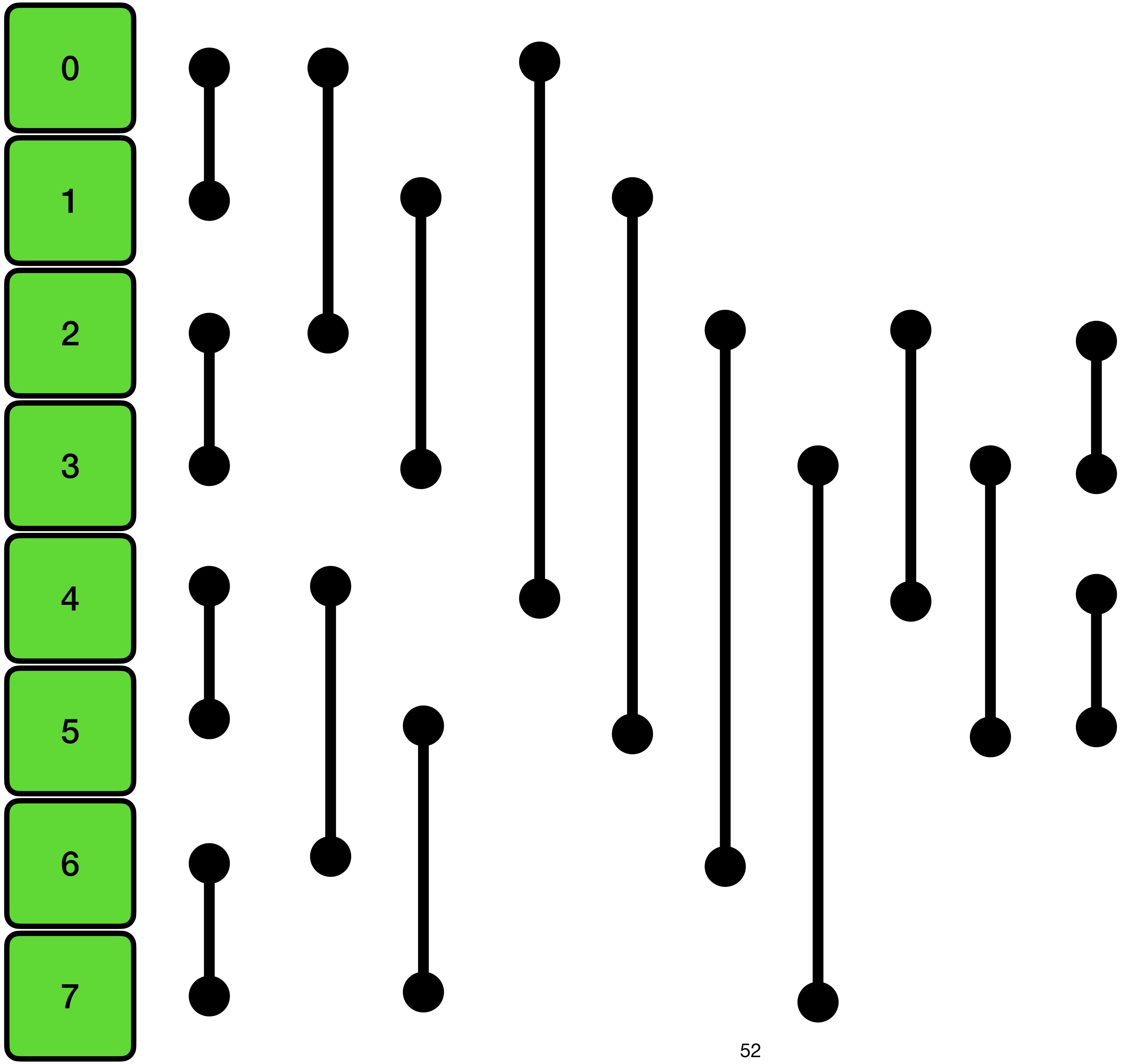
```
Ideal:
    $T \leftarrow \text{EmptyMap}$

    lookup($m$):
        if $m \notin T$:
            $T[m] \xleftarrow{\$} \{0,1\}^{\text{out}}$
        return $T[m]$
```

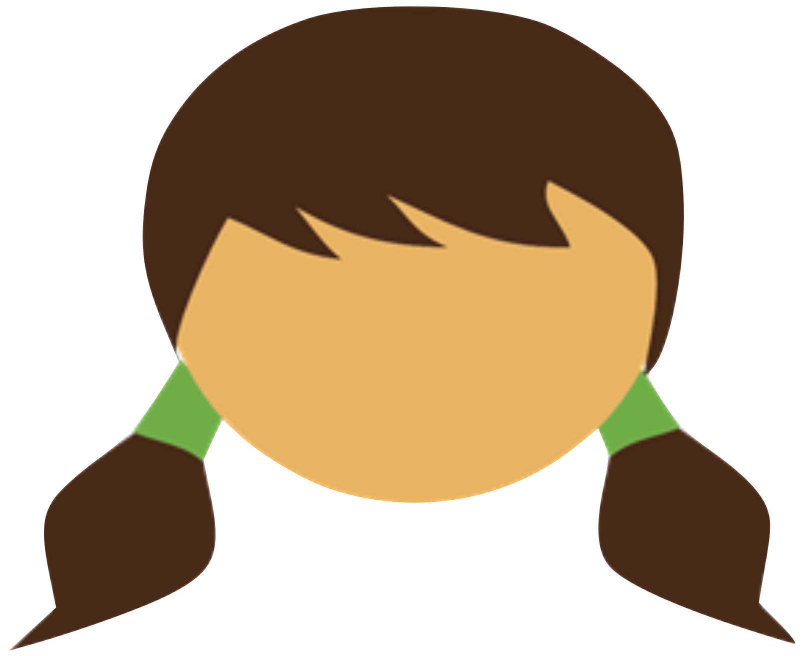
"F looks random"

How to shuffle



S

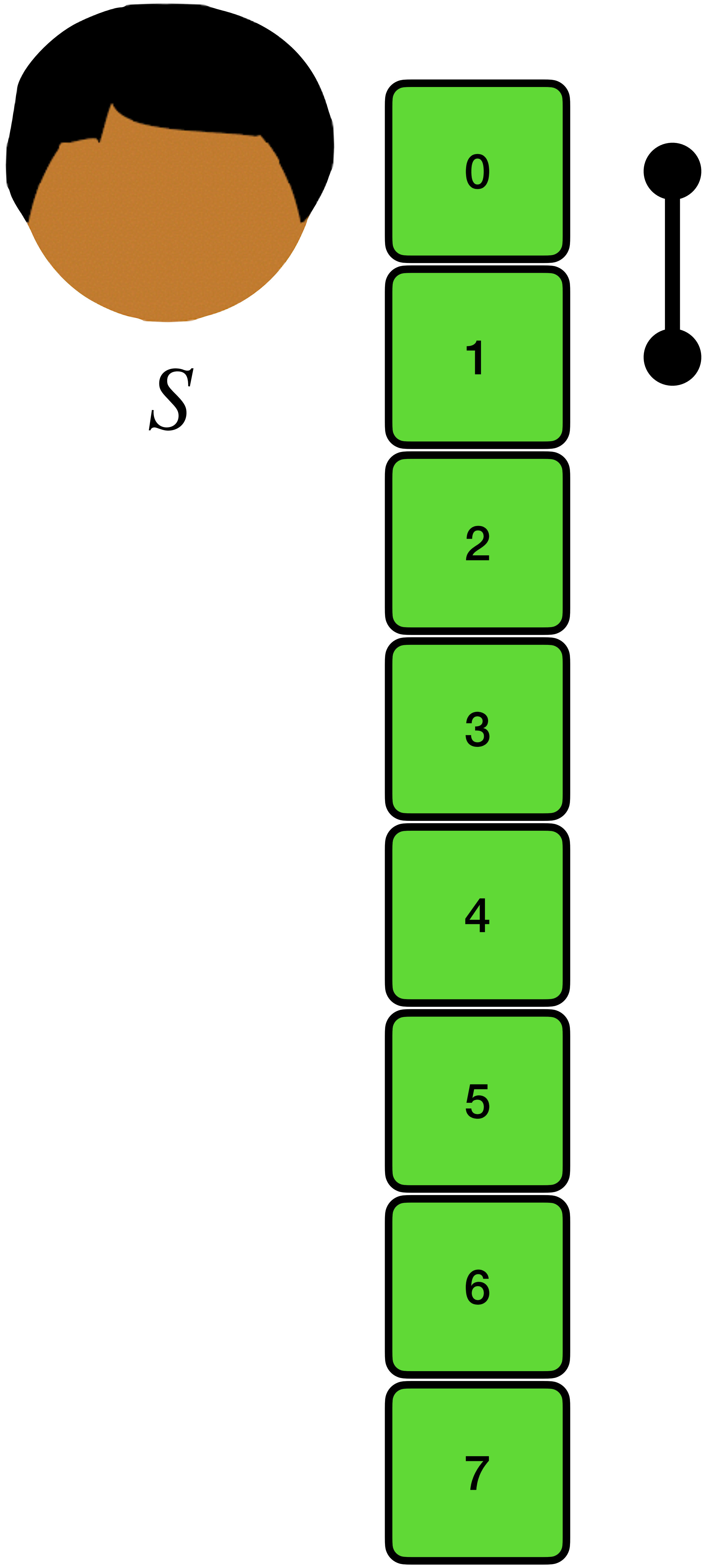


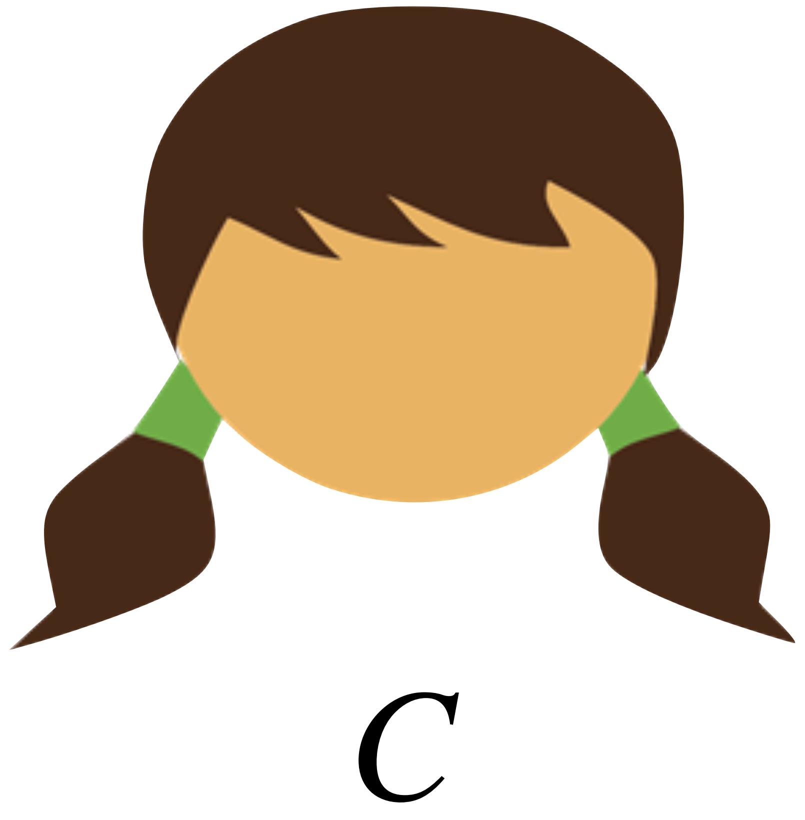


C

C samples a PRF key k_S

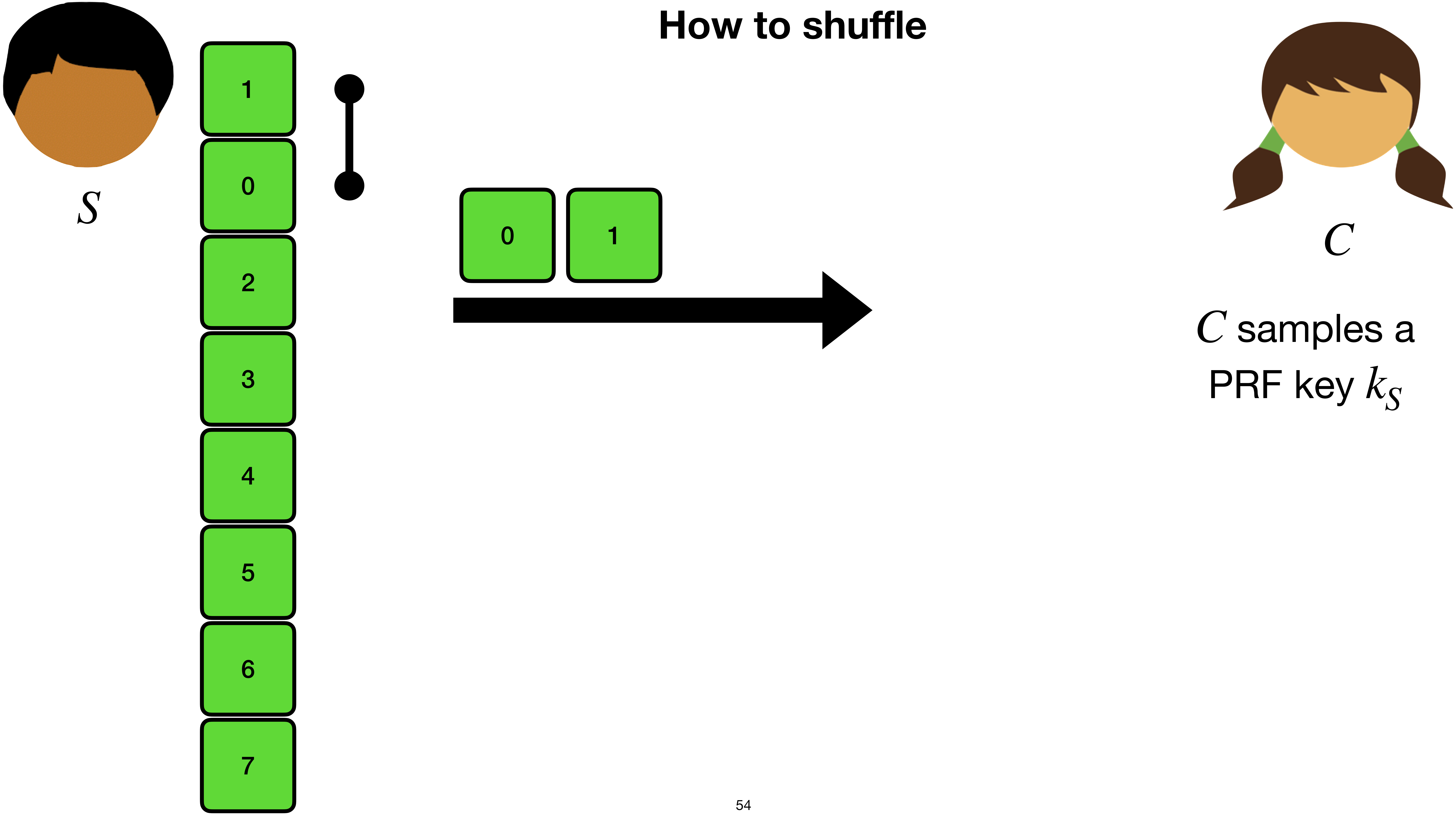
How to shuffle





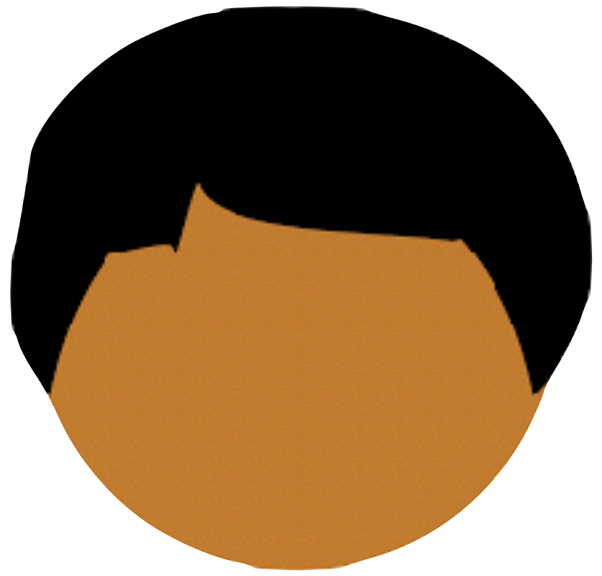
C samples a PRF key k_S

How to shuffle

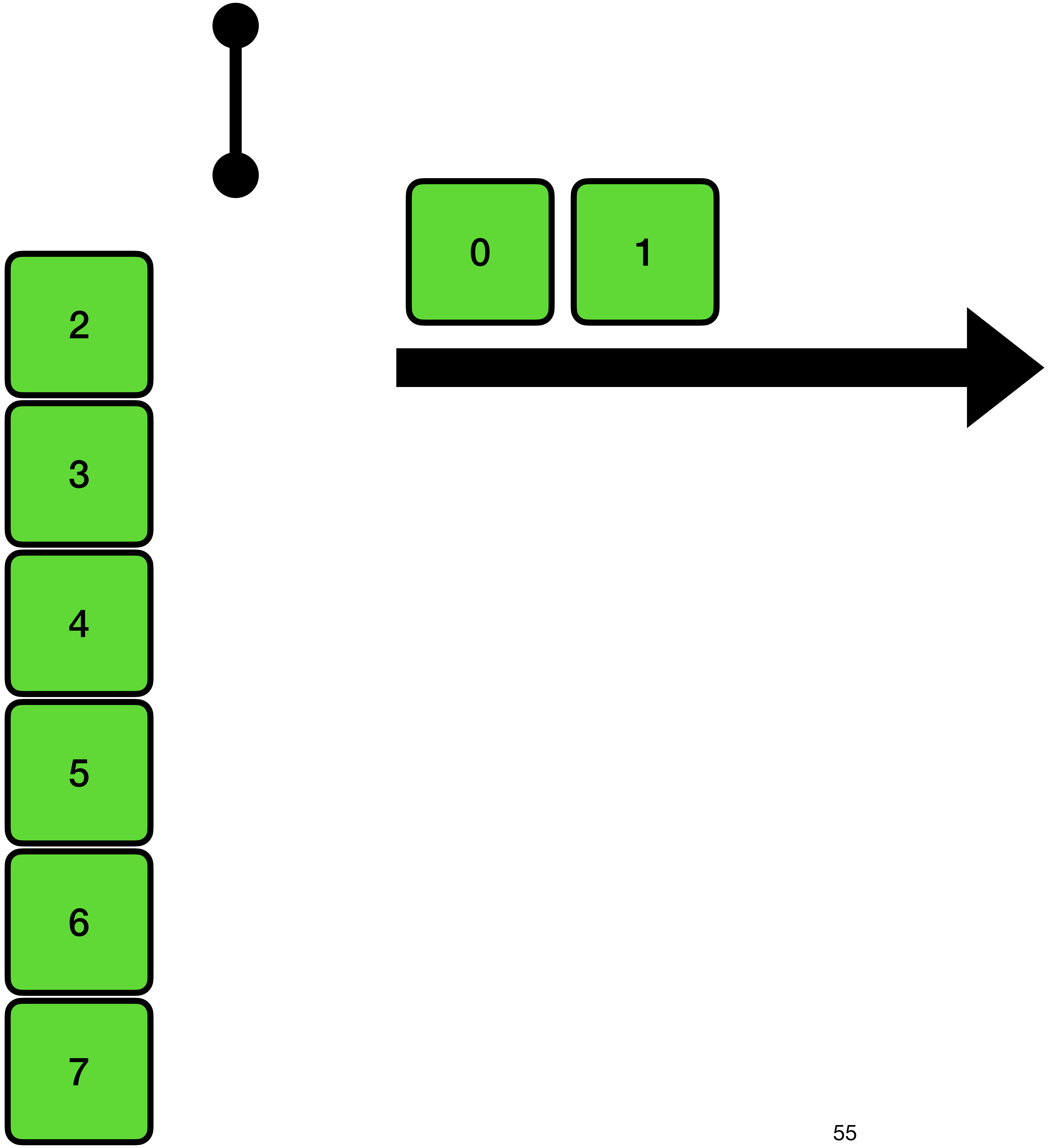


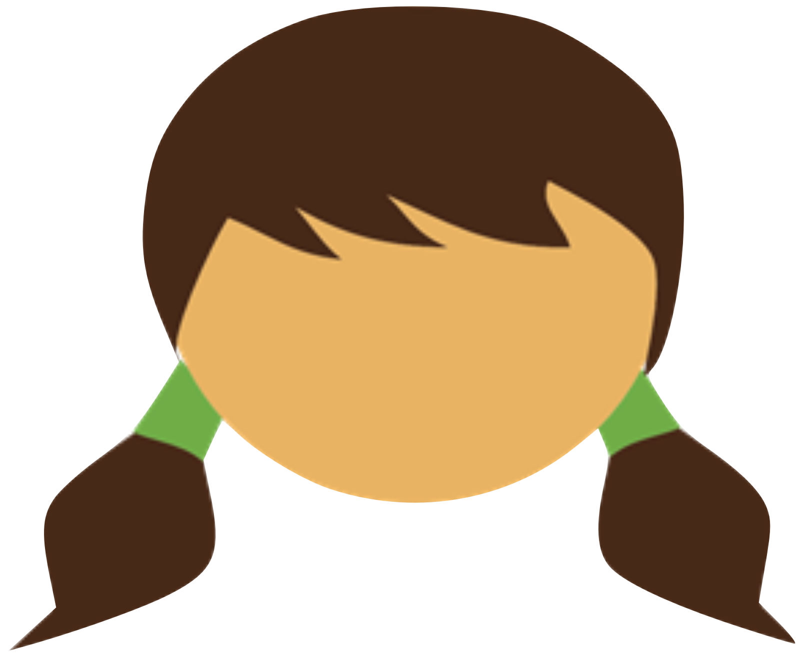
C samples a PRF key k_S

How to shuffle



S





C

C samples a PRF key k_S

$$F\left(K_s, \boxed{0}\right) \overset{?}{<} F\left(K_s, \boxed{1}\right)$$

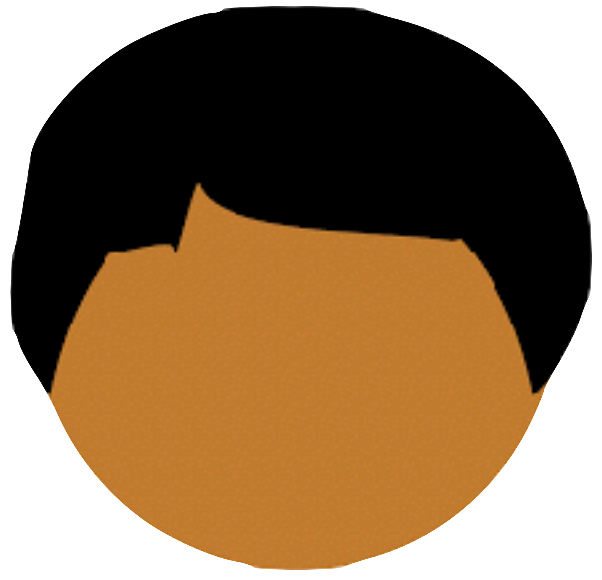
How to shuffle

S
2
3
4
5
6
7
0
1
0
1
1
0
C

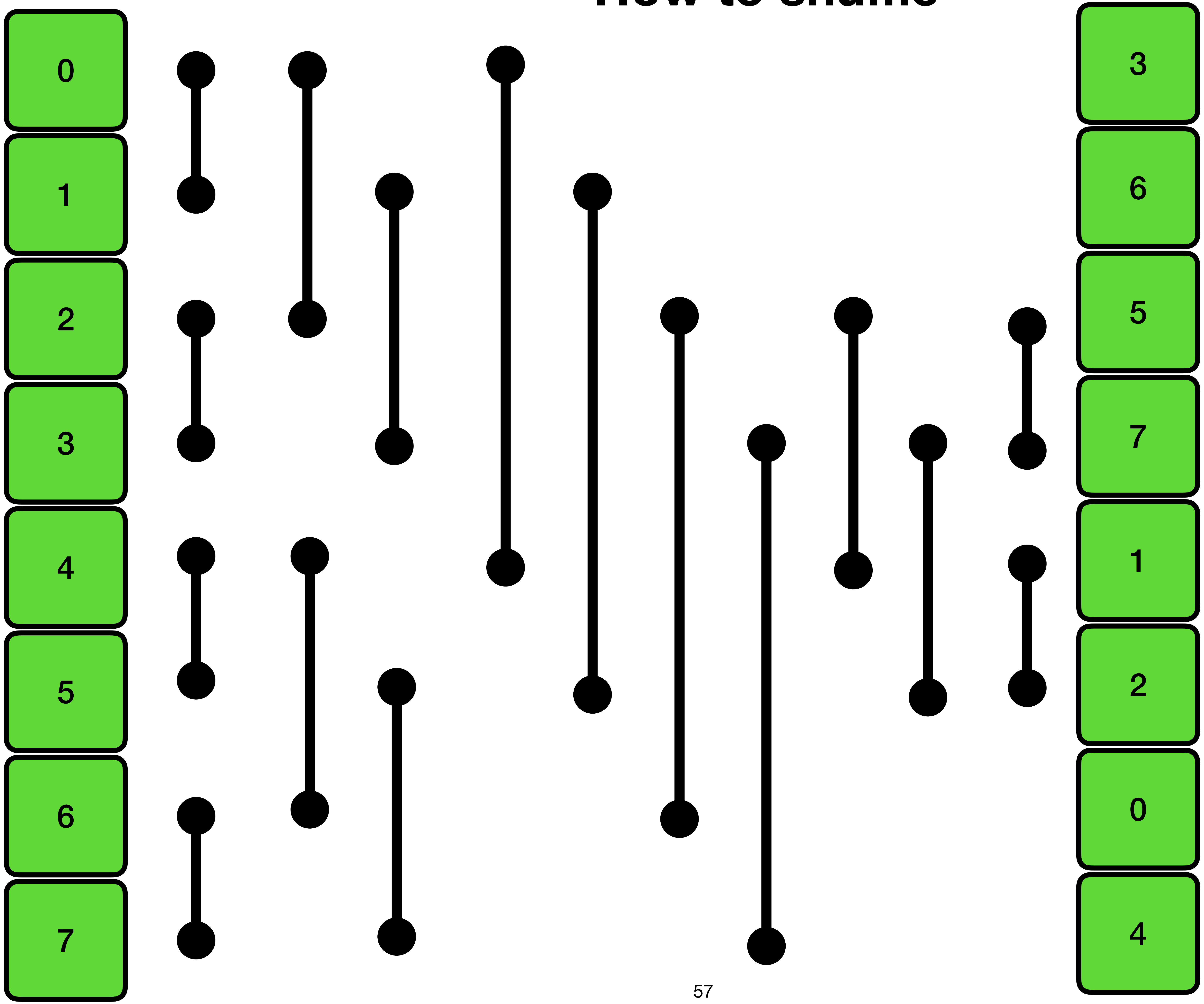
C samples a PRF key k_S

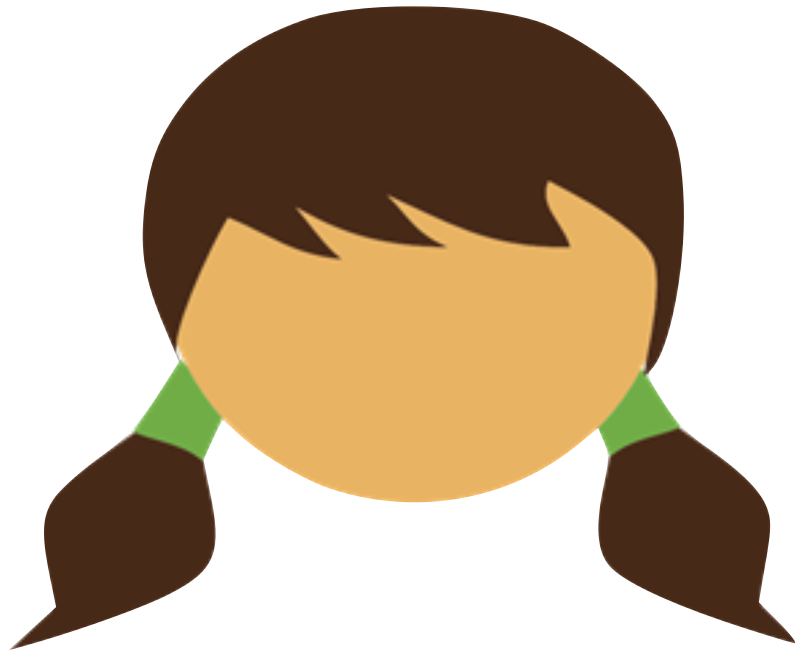
$$F\left(K_s, \boxed{0}\right) \overset{?}{<} F\left(K_s, \boxed{1}\right)$$

How to shuffle



S





C

C samples a PRF key k_S

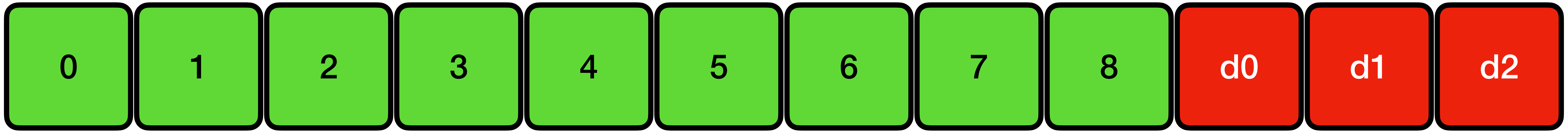
Square Root ORAM (Ostrovsky ’92)

Main idea: Shuffle all of RAM, but only roughly every $\sqrt{n}$ accesses

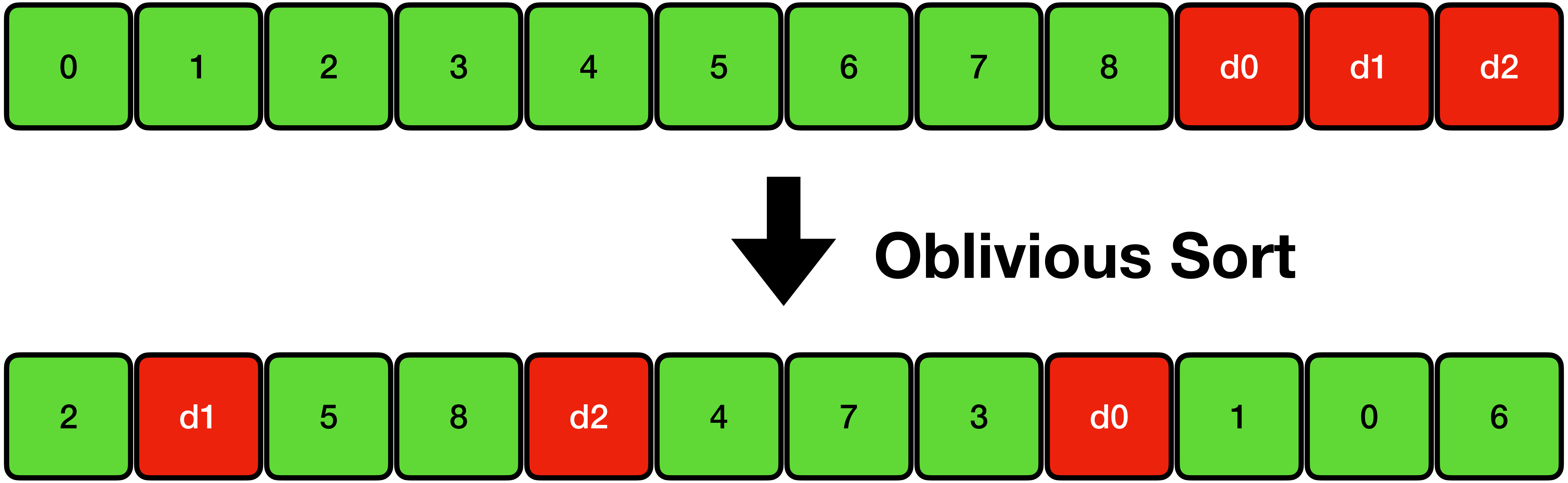
Square Root ORAM (Ostrovsky ’92)

0	1	2	3	4	5	6	7	8

Square Root ORAM (Ostrovsky ’92)



0	1	2	3	4	5	6	7	8	d0	d1	d2

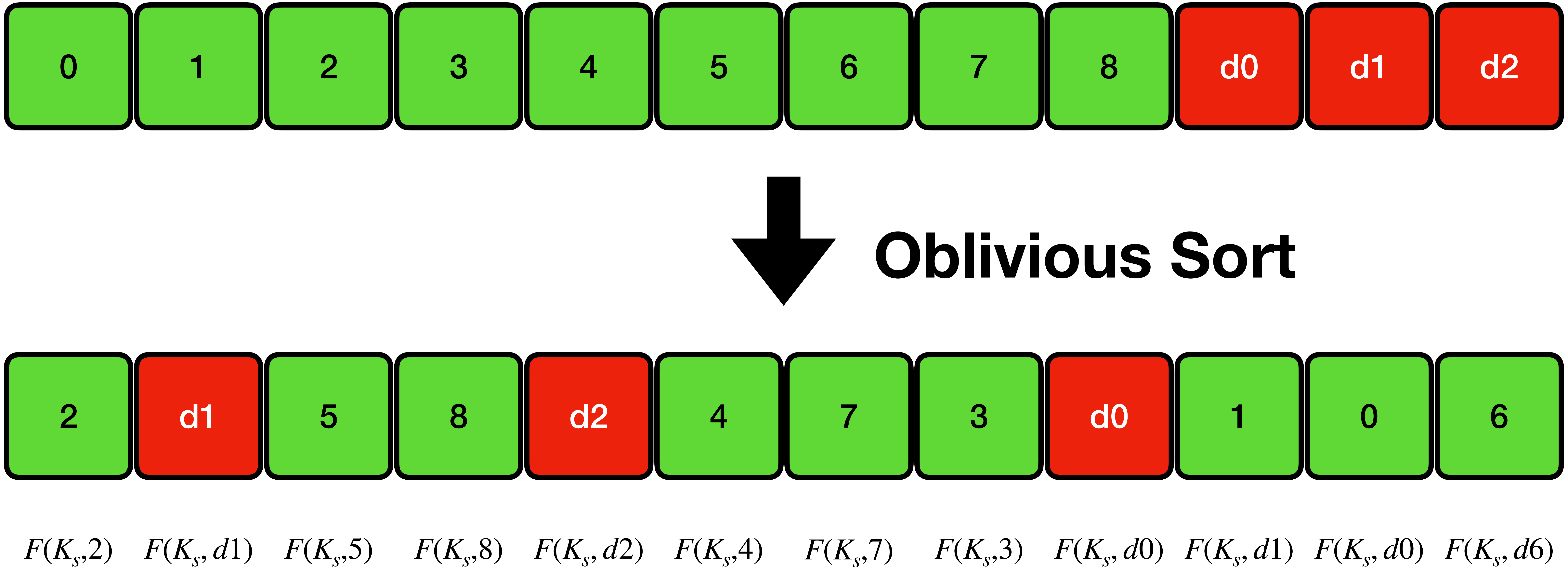


0 1 2 3 4 5 6 7 8 d0 d1 d2

Oblivious Sort

2 d1 5 8 d2 4 7 3 d0 1 0 6

Elements are sorted according to $F(k_S, r)$



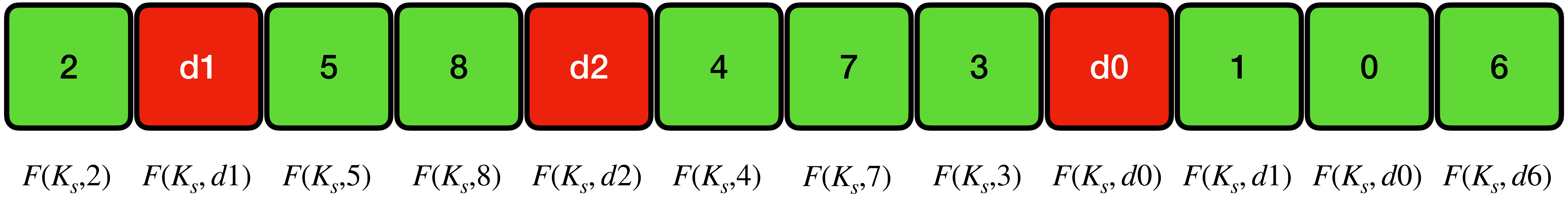
0	1	2	3	4	5	6	7	8	d0	d1	d2

Oblivious Sort

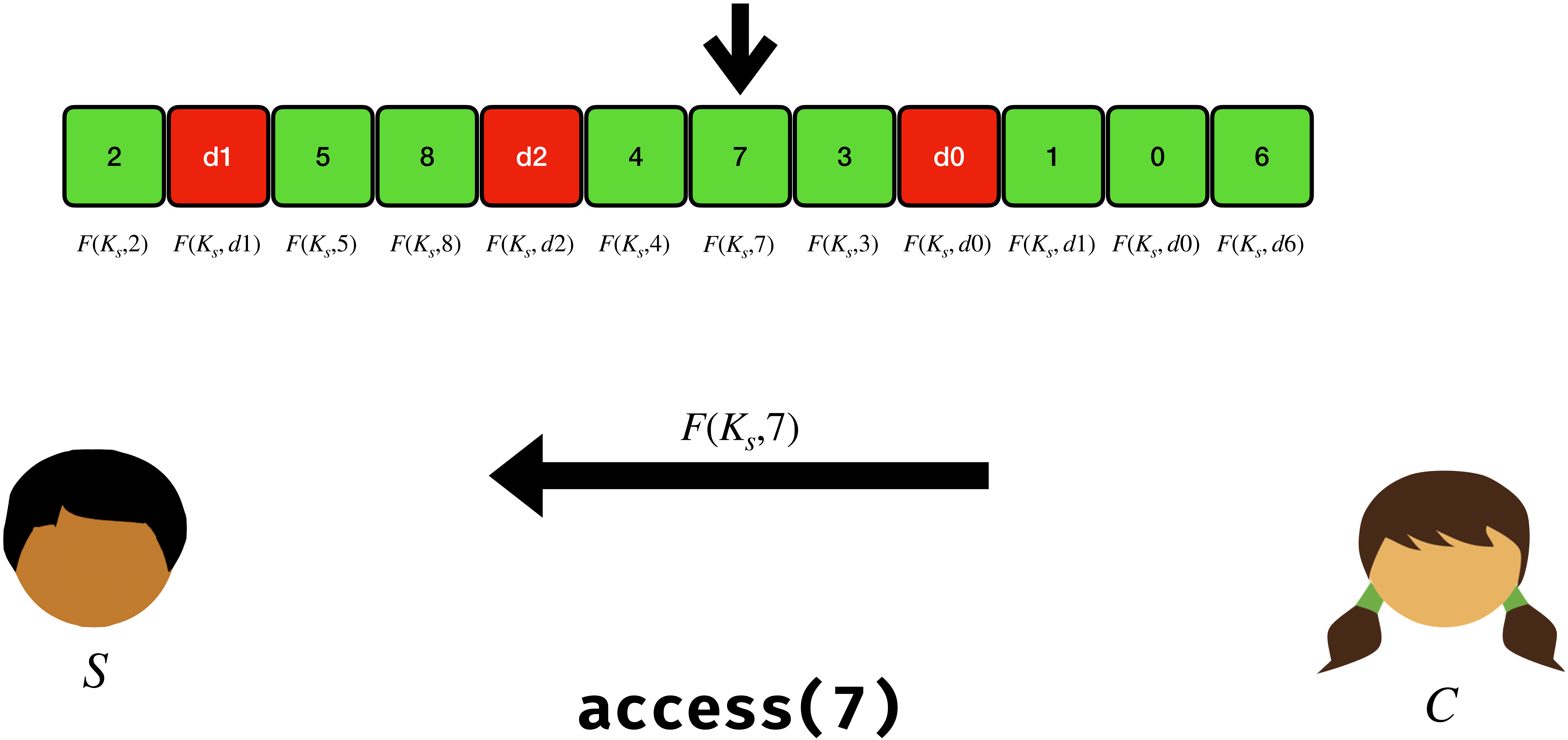
2	d1	5	8	d2	4	7	3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

Elements are sorted according to $F(k_S, \cdot\,)$

C then tags each element x with $F(k_S, x)$

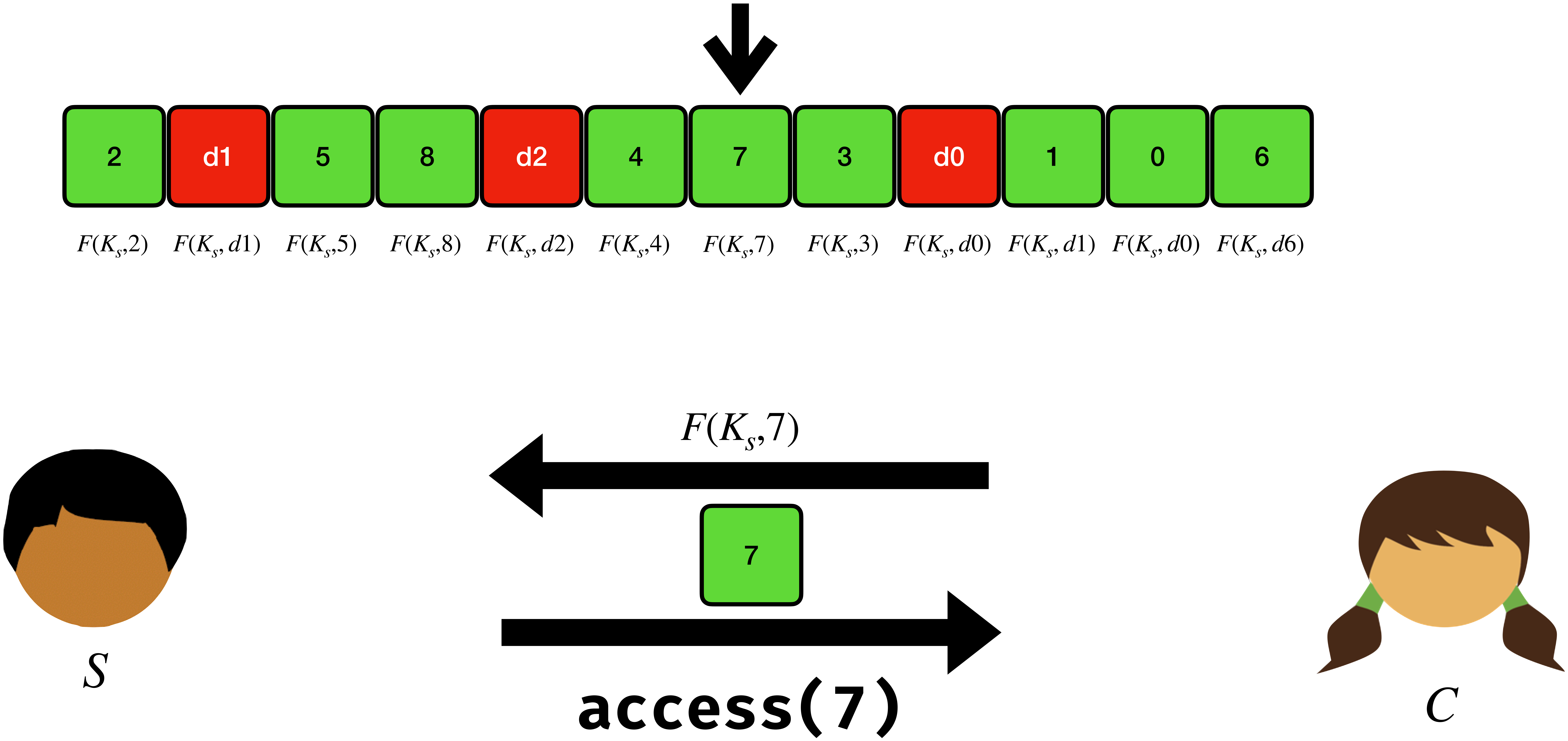


2	d1	5	8	d2	4	7	3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

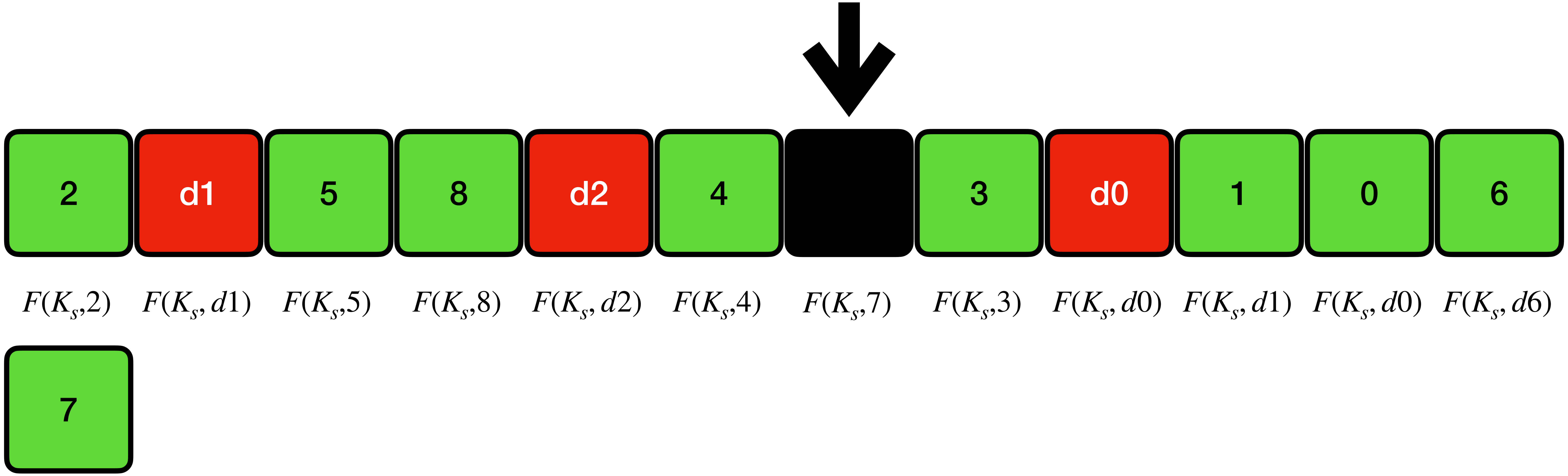


2	d1	5	8	d2	4	7	3	d0	1	0	6
$F(K_s,2)$	$F(K_s,d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s,d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s,d0)$	$F(K_s,d1)$	$F(K_s,d0)$	$F(K_s,d6)$

access(7)

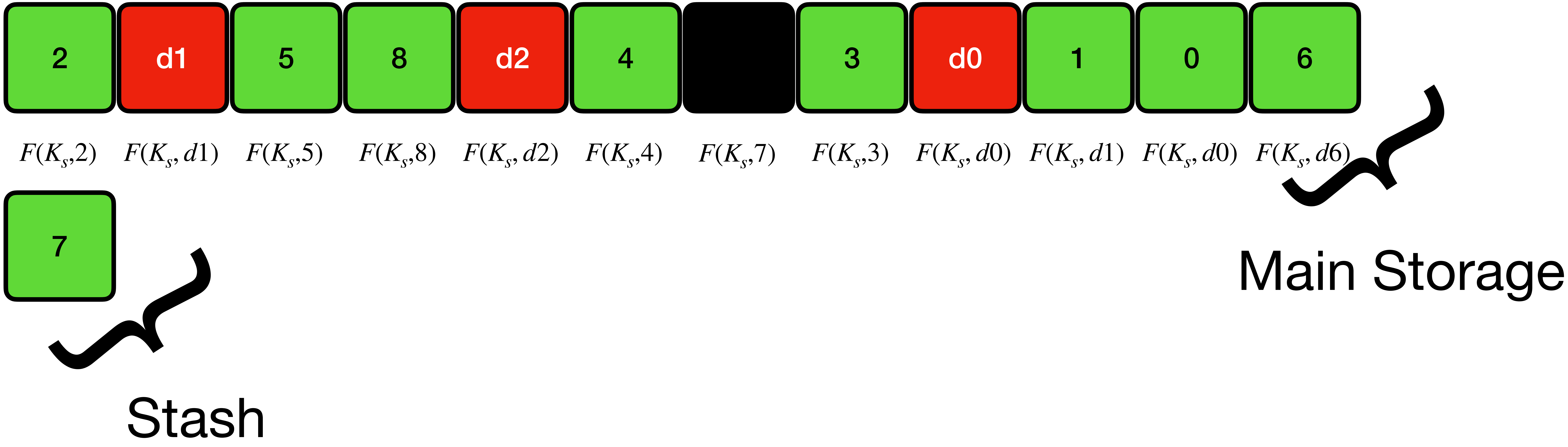


🟩 2	🟥 d1	🟩 5	🟩 8	🟥 d2	🟩 4	🟩 7	🟩 3	🟥 d0	🟩 1	🟩 0	🟩 6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$



2	d1	5	8	d2	4		3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

access(7)

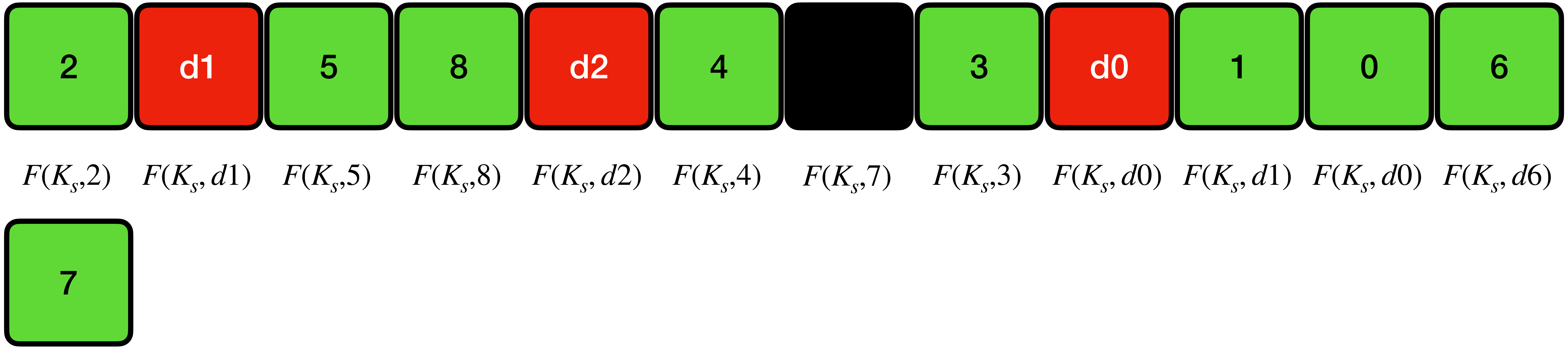


2	d1	5	8	d2	4		3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

7

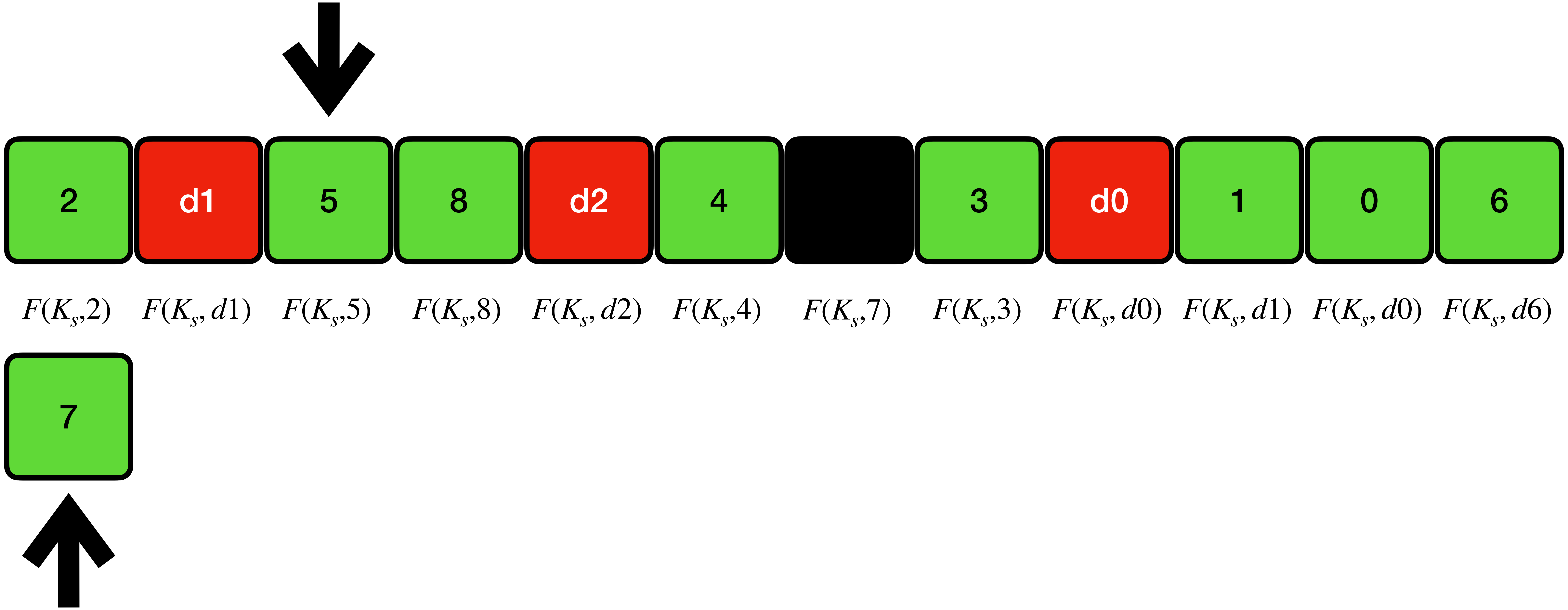
Main Storage

Stash



2	d1	5	8	d2	4		3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$
7											

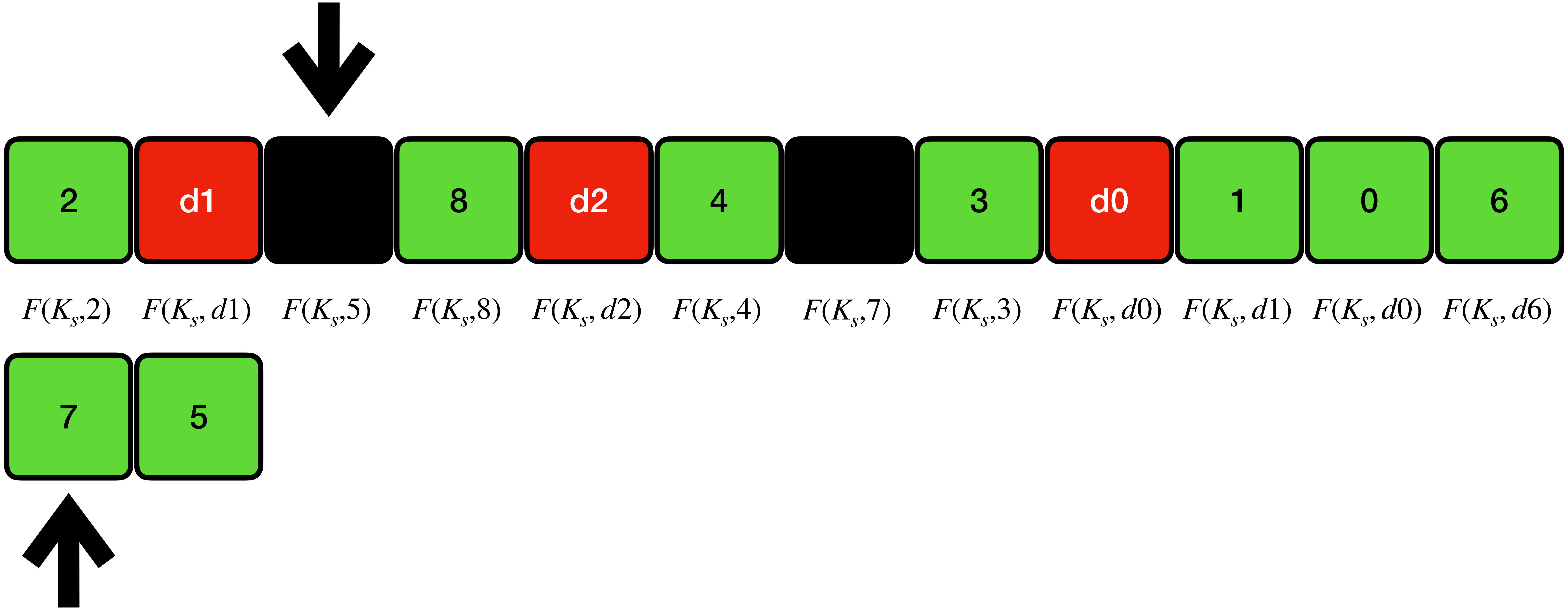
access(5)



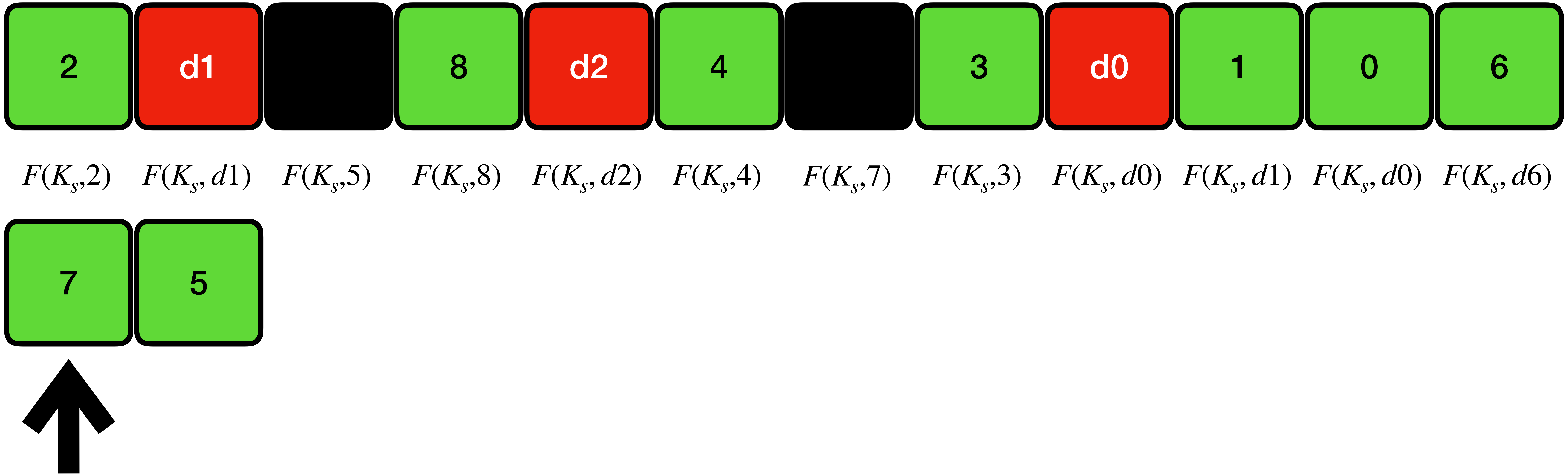
2	d1	5	8	d2	4		3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

7

access(5)

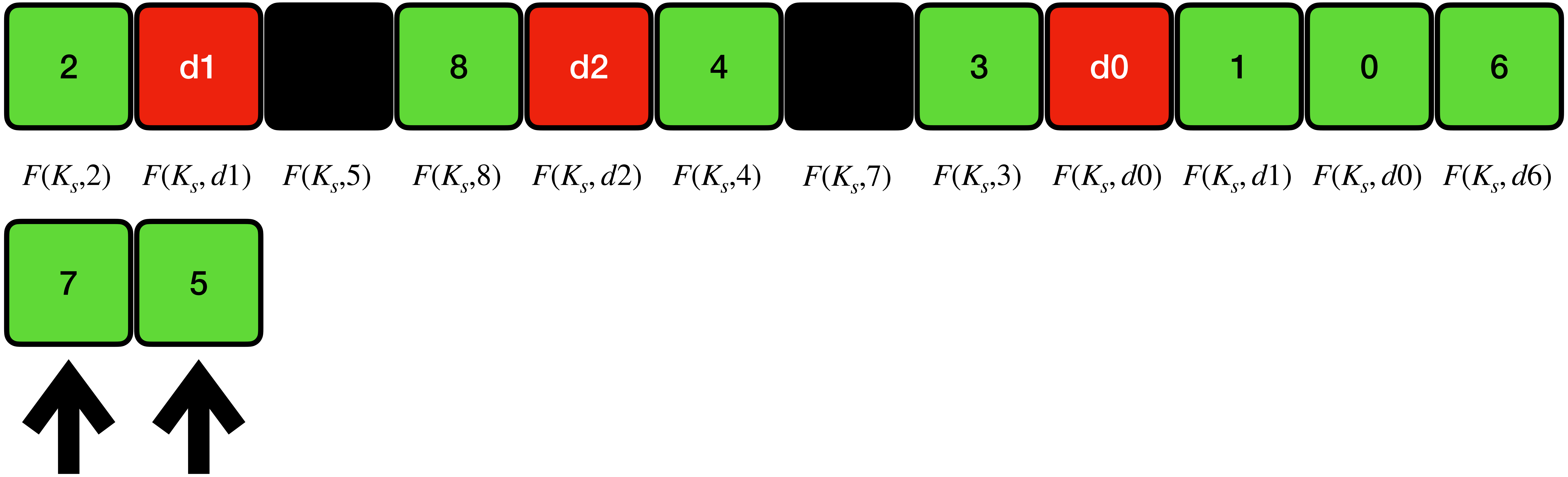

2
d1
8
d2
4
3
d0
1
0
6
$F(K_s,2)$
$F(K_s, d1)$
$F(K_s,5)$
$F(K_s,8)$
$F(K_s, d2)$
$F(K_s,4)$
$F(K_s,7)$
$F(K_s,3)$
$F(K_s, d0)$
$F(K_s, d1)$
$F(K_s, d0)$
$F(K_s, d6)$
7
5

access(5)



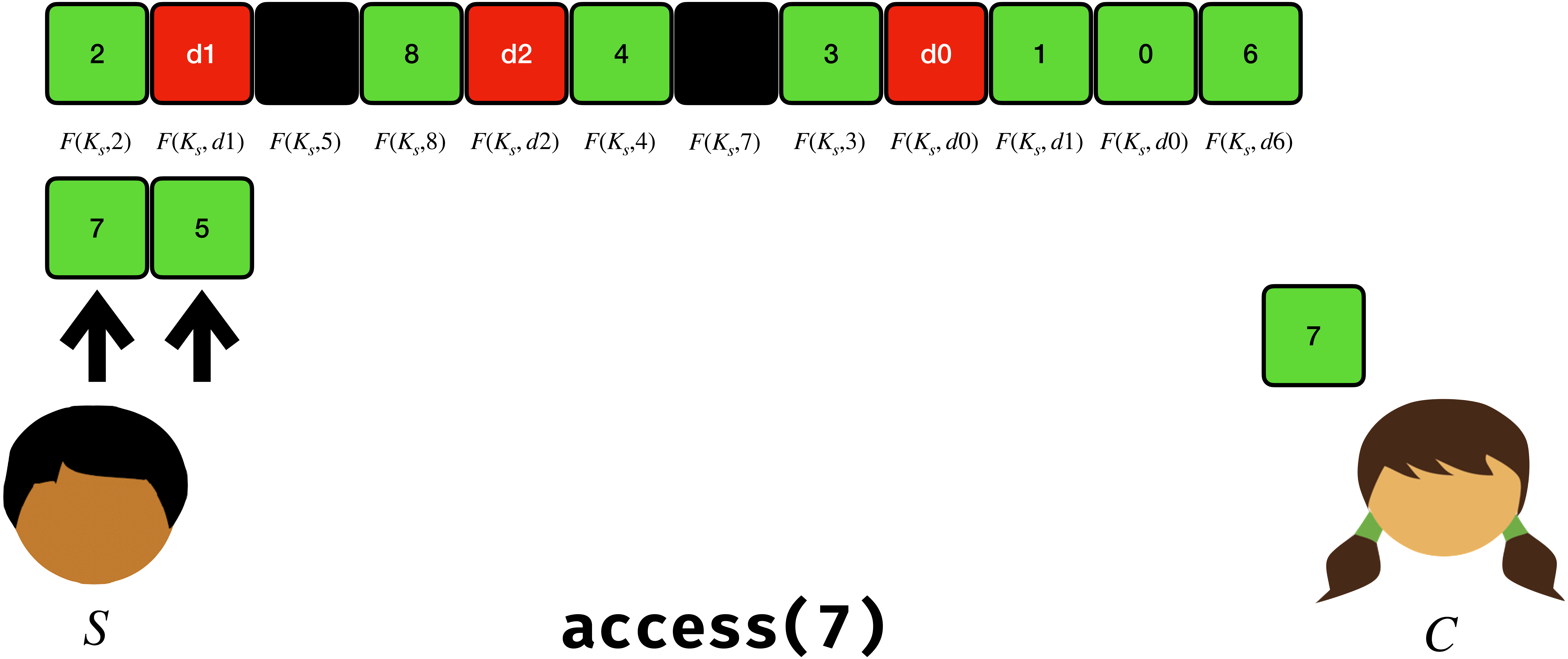
2	d1	■	8	d2	4	■	3	d0	1	0	6
$F(K_s,2)$	$F(K_s,d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s,d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s,d0)$	$F(K_s,d1)$	$F(K_s,d0)$	$F(K_s,d6)$
7	5										

access(7)



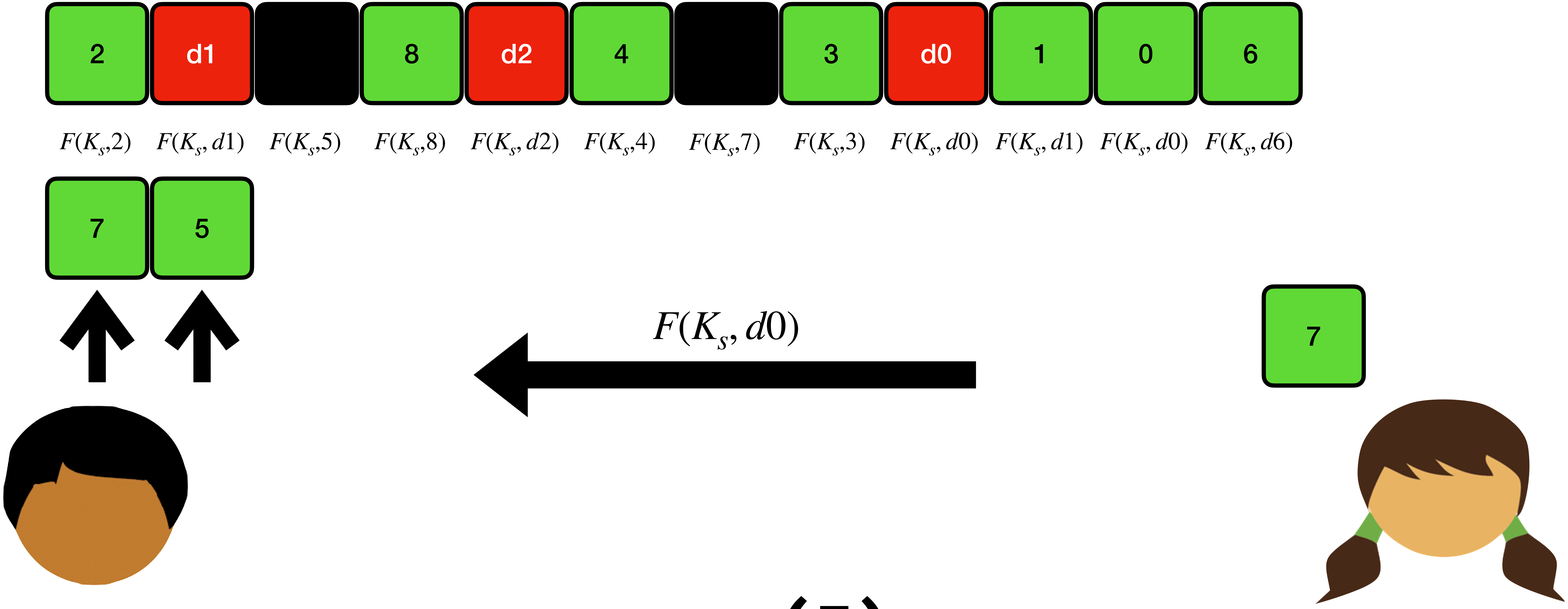
2	d1	■	8	d2	4	■	3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$
7	5										

access(7)



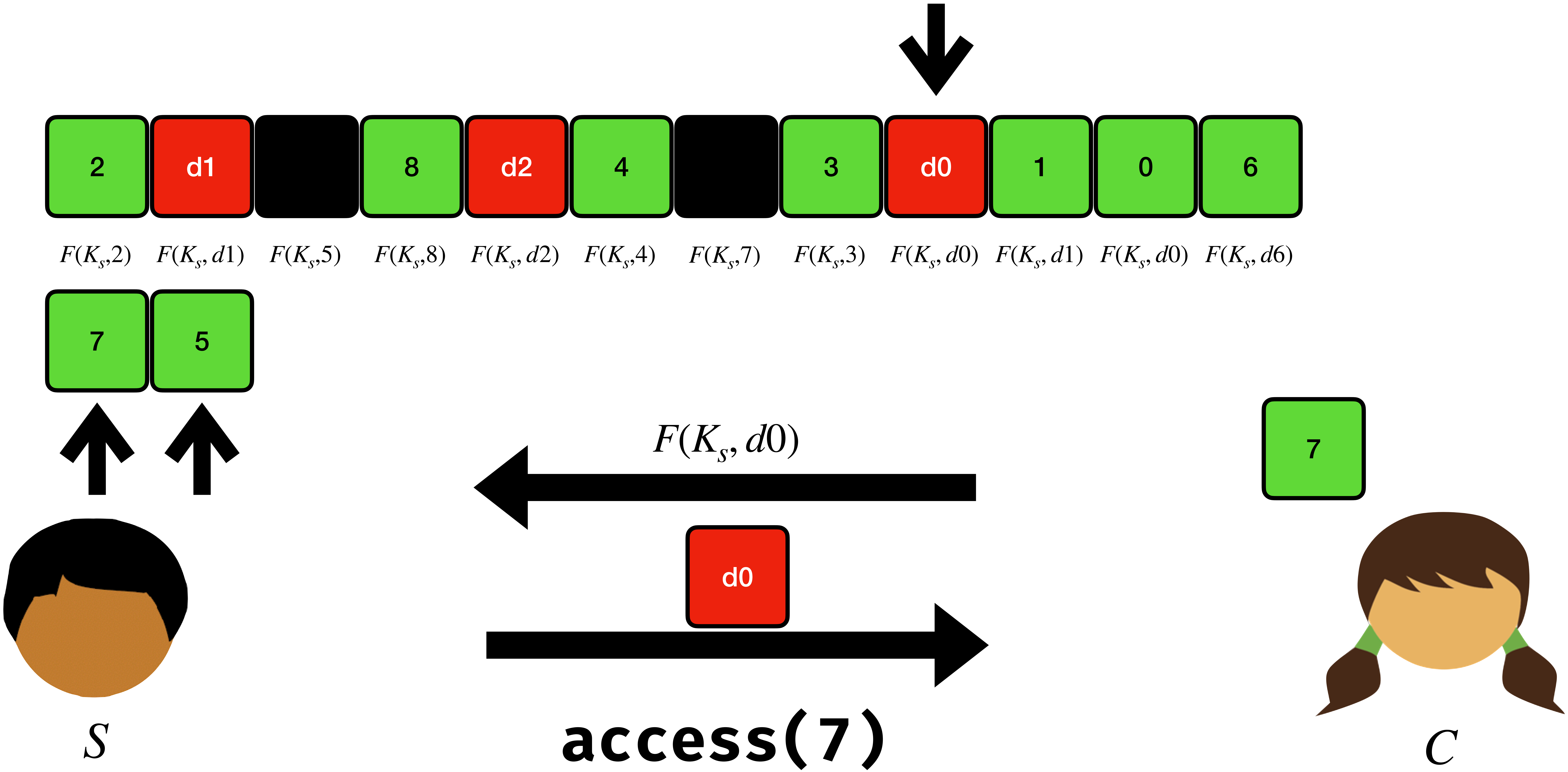
2	d1		8	d2	4		3	d0	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

access(7)



2
d1
8
d2
4
3
d0
1
0
6
$F(K_s,2)$
$F(K_s, d1)$
$F(K_s,5)$
$F(K_s,8)$
$F(K_s, d2)$
$F(K_s,4)$
$F(K_s,7)$
$F(K_s,3)$
$F(K_s, d0)$
$F(K_s, d1)$
$F(K_s, d0)$
$F(K_s, d6)$
7
5
$F(K_s, d0)$
7
S
C

access(7)



2 d1 8 d2 4 3 d0 1 0 6

$F(K_s,2)$ $F(K_s, d1)$ $F(K_s,5)$ $F(K_s,8)$ $F(K_s, d2)$ $F(K_s,4)$ $F(K_s,7)$ $F(K_s,3)$ $F(K_s, d0)$ $F(K_s, d1)$ $F(K_s, d0)$ $F(K_s, d6)$

7 5

$F(K_s, d0)$

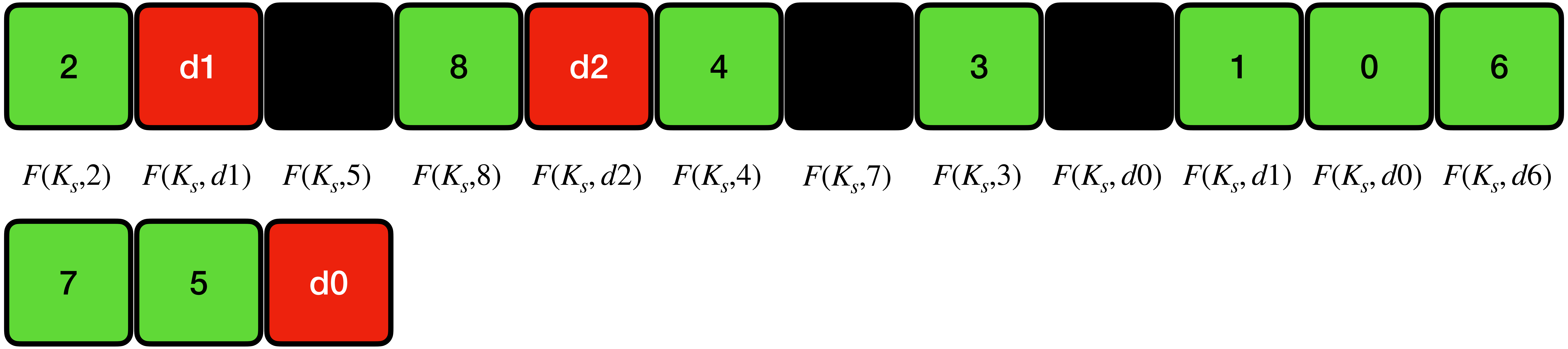
d0

7

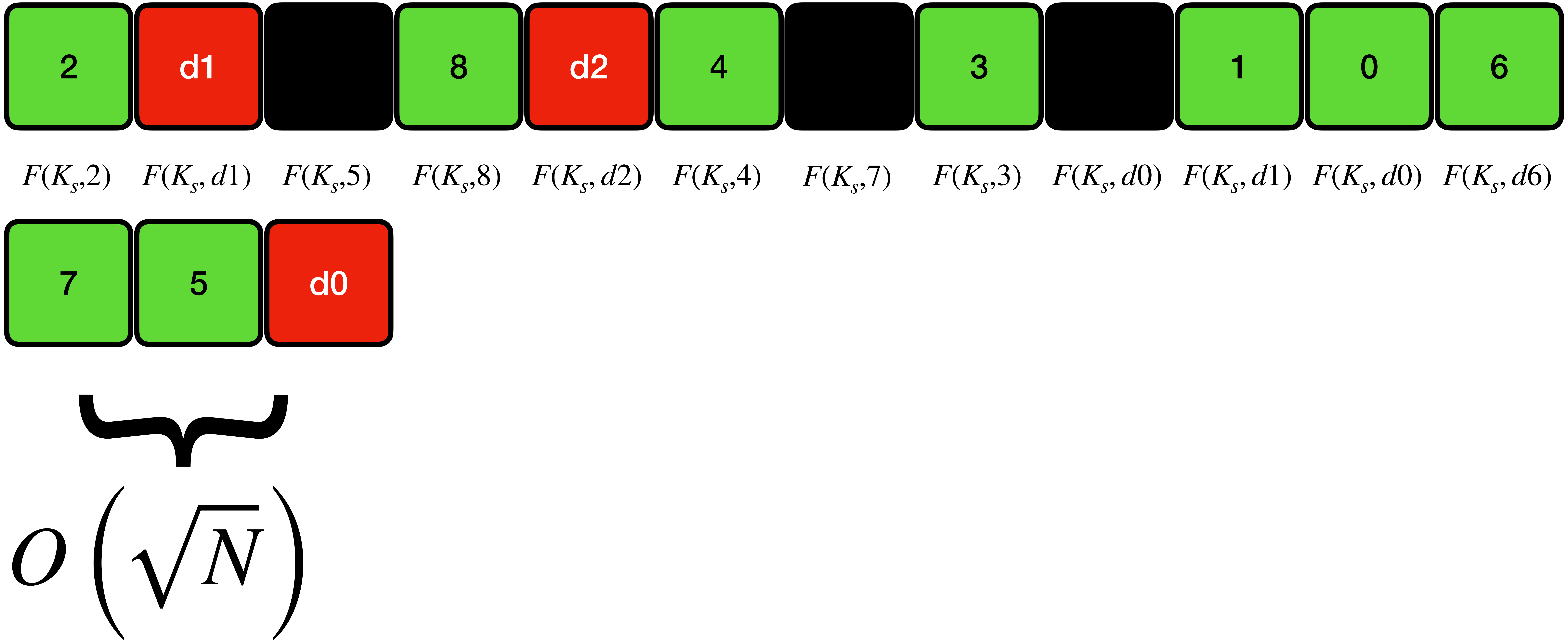
S

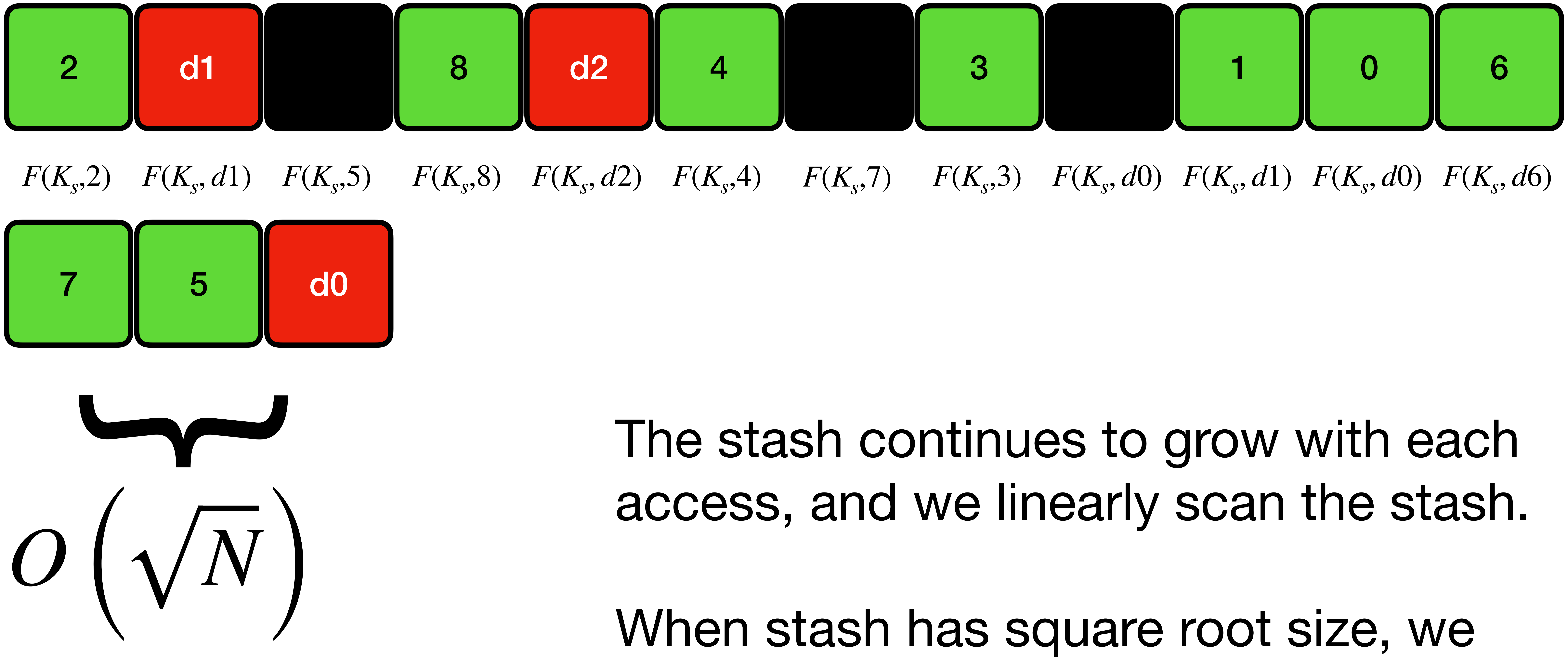
C

access(7)



2	d1	■	8	d2	4	■	3	■	1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$
7	5	d0									


2
d1
8
d2
4
3
1
0
6
$F(K_s,2)$
$F(K_s, d1)$
$F(K_s,5)$
$F(K_s,8)$
$F(K_s, d2)$
$F(K_s,4)$
$F(K_s,7)$
$F(K_s,3)$
$F(K_s, d0)$
$F(K_s, d1)$
$F(K_s, d0)$
$F(K_s, d6)$
7
5
d0
$O\left(\sqrt{N}\right)$



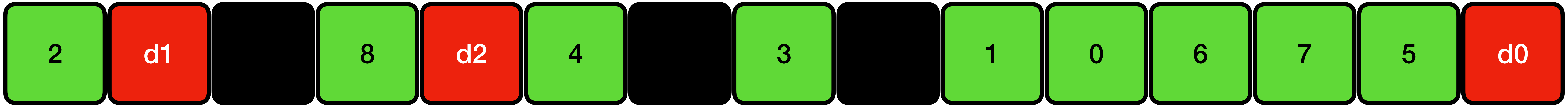
2	d1		8	d2	4		3		1	0	6
$F(K_s,2)$	$F(K_s, d1)$	$F(K_s,5)$	$F(K_s,8)$	$F(K_s, d2)$	$F(K_s,4)$	$F(K_s,7)$	$F(K_s,3)$	$F(K_s, d0)$	$F(K_s, d1)$	$F(K_s, d0)$	$F(K_s, d6)$

7	5	d0

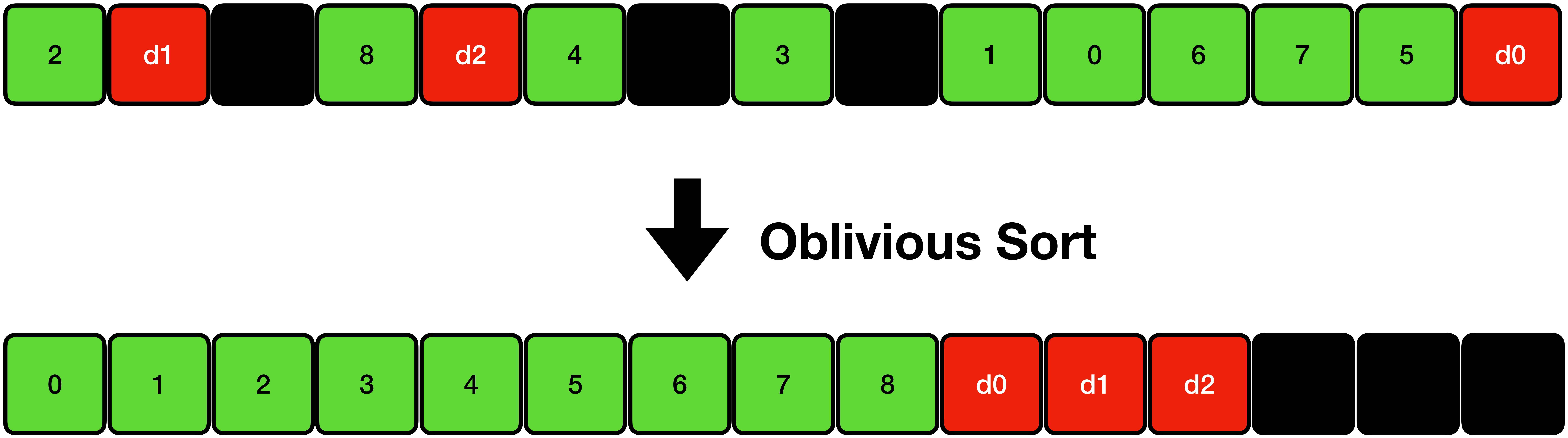
$$\underbrace{7 \quad 5}_{O\left(\sqrt{N}\right)}$$

The stash continues to grow with each access, and we linearly scan the stash.

When stash has square root size, we reset! (Reshuffle everything)



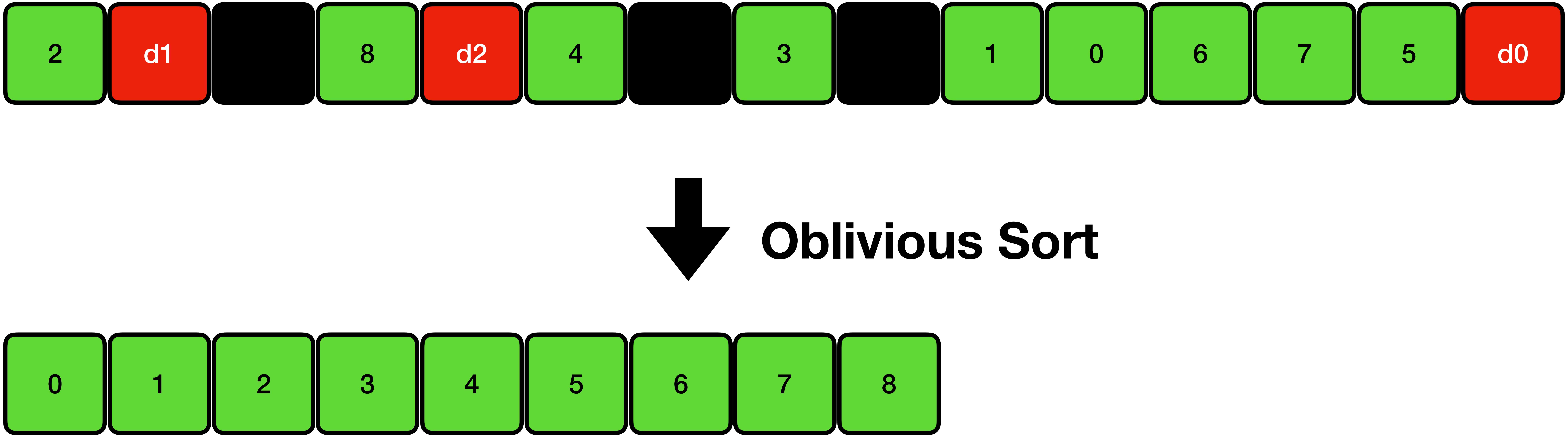
🟩 2	🟥 d1	⬛	🟩 8	🟥 d2	🟩 4	⬛	🟩 3	⬛	🟩 1	🟩 0	🟩 6	🟩 7	🟩 5	🟥 d0



2	d1	■	8	d2	4	■	3	■	1	0	6	7	5	d0

Oblivious Sort

0	1	2	3	4	5	6	7	8	d0	d1	d2	■	■	■



Oblivious Sort

0	1	2	3	4	5	6	7	8

Square Root ORAM (Ostrovsky ’92)

Main idea: Shuffle all of RAM, but only roughly every $\sqrt{n}$ accesses

Client shuffles via a sorting network (and by choosing ordering with a PRF key)

On each access, linearly scan the stash

- If the element is not in the stash, client directly asks for element from main storage

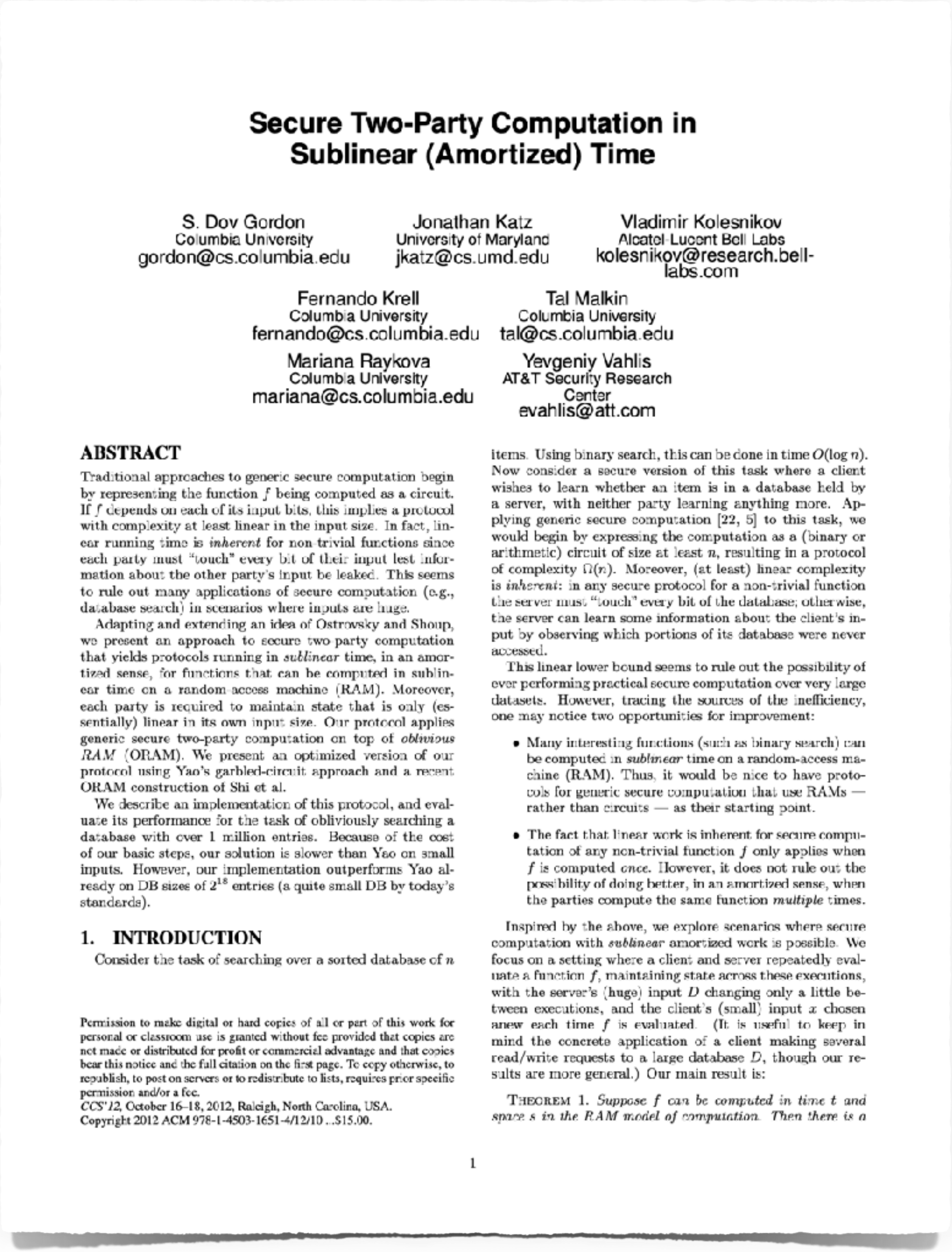
- Otherwise, client asks for a dummy from main storage (client keeps a dummy counter to always ask for fresh dummies)

Every $O(\sqrt{n})$ accesses, reshuffle to keep the stash size in check

Square Root ORAM (Ostrovsky ’92)

Security: on each access, server linearly scans stash, sees a request for a uniformly random element (without replacement) from main storage

Efficiency: Roughly $\sqrt{n}$ physical accesses per logical access



Secure Two-Party Computation in Sublinear (Amortized) Time

S. Dov Gordon
Columbia University
gordon@cs.columbia.edu

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

Vladimir Kolesnikov
Alcatel-Lucent Bell Labs
kolesnikov@research.bell-labs.com

Fernando Krell
Columbia University
fernando@cs.columbia.edu

Tal Malkin
Columbia University
tal@cs.columbia.edu

Mariana Raykova
Columbia University
mariana@cs.columbia.edu

Yevgeniy Vahlis
AT&T Security Research Center
evahlis@att.com

ABSTRACT

Traditional approaches to generic secure computation begin by representing the function f being computed as a circuit. If f depends on each of its input bits, this implies a protocol with complexity at least linear in the input size. In fact, linear running time is *inherent* for non-trivial functions since each party must "touch" every bit of their input lest information about the other party's input be leaked. This seems to rule out many applications of secure computation (e.g., database search) in scenarios where inputs are huge.

Adapting and extending an idea of Ostrovsky and Shoup, we present an approach to secure two-party computation that yields protocols running in *sublinear* time, in an amortized sense, for functions that can be computed in sublinear time on a random-access machine (RAM). Moreover, each party is required to maintain state that is only (essentially) linear in its own input size. Our protocol applies generic secure two-party computation on top of *oblivious RAM* (ORAM). We present an optimized version of our protocol using Yao's garbled-circuit approach and a recent ORAM construction of Shi et al.

We describe an implementation of this protocol, and evaluate its performance for the task of obliviously searching a database with over 1 million entries. Because of the cost of our basic steps, our solution is slower than Yao on small inputs. However, our implementation outperforms Yao already on DB sizes of 2^{18} entries (a quite small DB by today's standards).

1. INTRODUCTION

Consider the task of searching over a sorted database of n items. Using binary search, this can be done in time $O(\log n)$. Now consider a secure version of this task where a client wishes to learn whether an item is in a database held by a server, with neither party learning anything more. Applying generic secure computation [22, 5] to this task, we would begin by expressing the computation as a (binary or arithmetic) circuit of size at least n, resulting in a protocol of complexity $\Omega(n)$. Moreover, (at least) linear complexity is *inherent*: in any secure protocol for a non-trivial function the server must "touch" every bit of the database; otherwise, the server can learn some information about the client's input by observing which portions of its database were never accessed.

This linear lower bound seems to rule out the possibility of ever performing practical secure computation over very large datasets. However, tracing the sources of the inefficiency, one may notice two opportunities for improvement:

- Many interesting functions (such as binary search) can be computed in *sublinear* time on a random-access machine (RAM). Thus, it would be nice to have protocols for generic secure computation that use RAMs — rather than circuits — as their starting point.

- The fact that linear work is inherent for secure computation of any non-trivial function f only applies when f is computed *once*. However, it does not rule out the possibility of doing better, in an amortized sense, when the parties compute the same function *multiple* times.

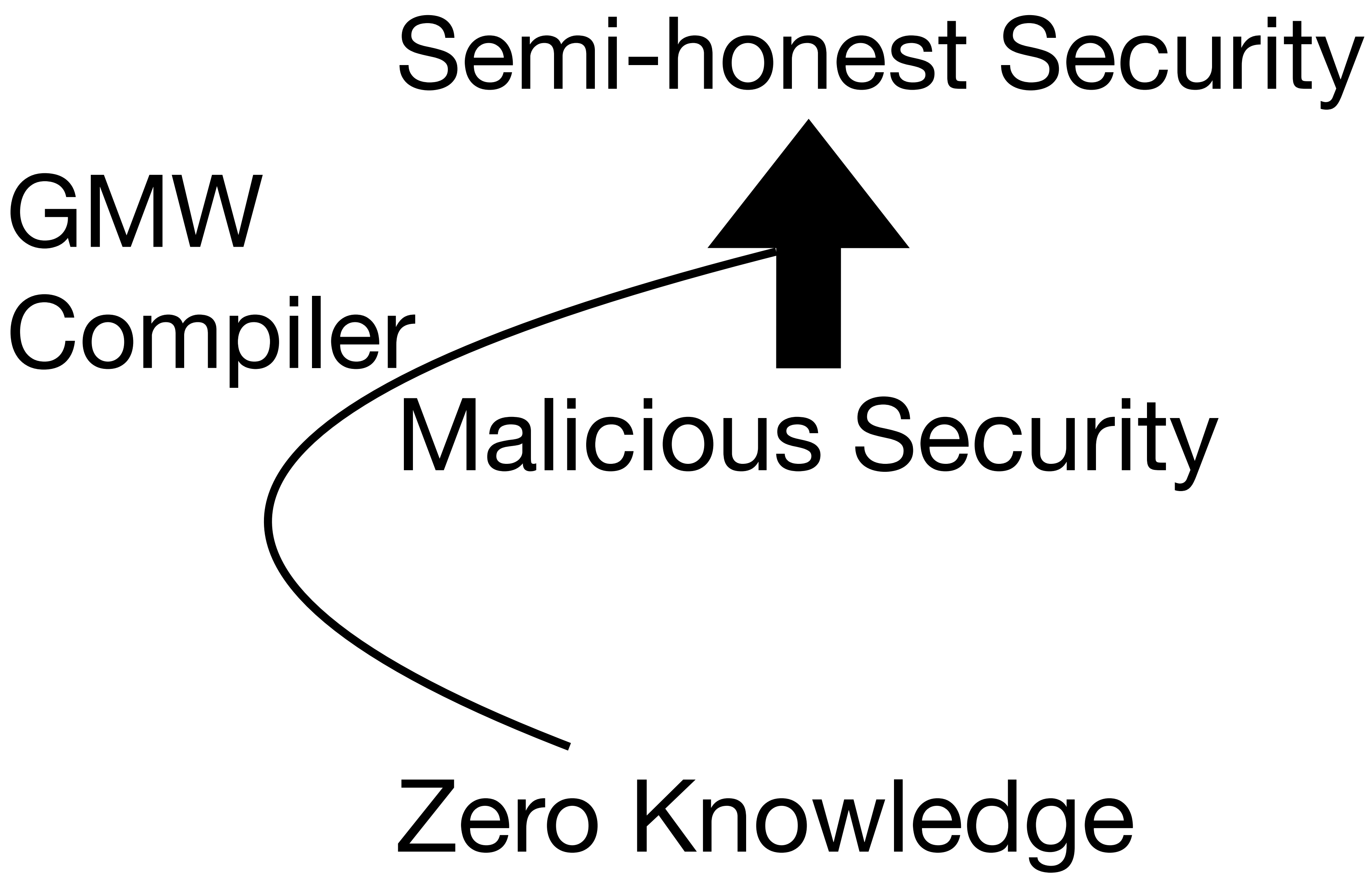
Inspired by the above, we explore scenarios where secure computation with *sublinear* amortized work is possible. We focus on a setting where a client and server repeatedly evaluate a function f, maintaining state across these executions, with the server's (huge) input D changing only a little between executions, and the client's (small) input x chosen anew each time f is evaluated. (It is useful to keep in mind the concrete application of a client making several read/write requests to a large database D, though our results are more general.) Our main result is:

THEOREM 1. *Suppose f can be computed in time t and space s in the RAM model of computation. Then there is a*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

Setting



Semi-honest Security

GMW Compiler

Malicious Security

Zero Knowledge

General-Purpose Tools

GMW Protocol

- Multi-party
- Multi-round

Garbled Circuit

- Constant Round
- Two Party

Primitives

Oblivious Transfer

Pseudorandom functions/encryption

Commitments

ORAM

Today’s objectives

Introduce Oblivious RAM (ORAM)

Define ORAM Security

Construct non-trivial ORAM

Discuss how ORAM can be plugged into MPC